mimik Technology White Paper

App-Native vs. Agentix-Native Architectures for On-Device AI Agents

A Quantitative Engineering Guide

Using Device-First Continuum AI

Siavash M. Alamouti   Fay Arjomandi   Michel Burger   Jeremy Hsu

mimik Technology Inc.

developer.mimik.com

Executive Summary

As on-device AI agents proliferate across domains, including continuous health monitoring, industrial IoT, smart infrastructure, and personal assistants, a fundamental architectural question faces every development team: should agents be organized within the constraints of app-native service mechanisms, which in practice produce monolithic architectures with tightly coupled deployment and failure domains, or should they be built as independently deployable microservice-based agents within a purpose-built agent operating environment?

We term these two paradigms *app-native* and *Agentix-native*. In the app-native approach, developers build agents within a single application using platform IPC mechanisms (Android Bound Services, iOS XPC, Linux D-Bus), which lack the primitives required for true agent independence and force monolithic coupling. In the Agentix-native approach, each agent is packaged as a *mim* (micro intelligence module), a serverless microservice running on a lightweight operating environment (*mimOE*) that provides service discovery, lifecycle management, and continuum coordination.

This white paper provides a systematic engineering comparison of these two paradigms within the Device-First Continuum AI (DFC-AI) framework. Using analytical models parameterized entirely by published benchmarks, we evaluate tradeoffs across seven dimensions: evolvability, composability, inter-agent reasoning, regulatory modularity, continuum mobility, operational overhead, and system resilience.

Our analysis incorporates a serverless microservice-based agent architecture in which each mim operates as a short-lived serverless process with independent lifecycle management, while model weights are maintained in a shared registry with LRU caching and deduplication across agents that share backbone models.

The results show that while app-native monolithic agents offer marginal raw efficiency advantages (1.2–2× per inference), the Agentix-native microservice architecture delivers substantial benefits: 97% reduction in update payloads (99.4% for code-only updates), 59% storage savings through dynamic composition with model deduplication, 71% reduction in regulatory verification effort, 73% reduction in multi-platform engineering effort, and a 7.8× reduction in failure blast radius. The serverless memory model yields expected memory usage 30% *lower* than the monolithic architecture at eight agents, because low-frequency agents consume zero resources between invocations. More fundamentally, the Agentix-native architecture is *necessary* for any agent that must communicate beyond its own application process. Since real-world deployments inherently require such communication, the agent operating environment is not an optimization but a prerequisite.

Contents

1 The Multi-Agent Problem 3

2 Device-First Continuum AI 3

3 Architecture Definitions 5
3.1 Common Ground 5
3.2 App-Native Architecture (Monolithic) 5
3.3 Agentix-Native Architecture (Microservice-Based) 5

4 Why App-Native Mechanisms Force Monolithic Coupling 7
4.1 Android Bound Services and AIDL 7
4.2 iOS XPC Services and App Extensions 8
4.3 Linux D-Bus 8
4.4 ROS 2 and DDS 9
4.5 Summary: From App-Native Constraints to Agentix-Native Primitives 9

5 Quantitative Analysis Framework 10
5.1 Parameterization from Published Benchmarks 10
5.2 Reference Workload: Health Monitoring 11
5.3 Energy Model 11
5.4 Serverless Memory Model 12
5.5 Simulation Scenarios 12

6 Results: Seven Dimensions Compared 13
6.1 Dimension 1: Evolvability 13
6.2 Dimension 2: Composability 13
6.3 Dimension 3: Inter-Agent Reasoning 14
6.4 Dimension 4: Regulatory Modularity 14
6.5 Dimension 5: Continuum Mobility 15
6.6 Dimension 6: Operational Overhead 15
6.7 Dimension 7: System Resilience 17

7 Sensitivity Analysis 17

8 Consolidated Comparison 19

9 Why the App-Native Assumption Fails in Practice 19
9.1 The Isolation Fallacy 19
9.2 Architectural Discontinuity 22
9.3 No Crossover Point 22

10 Beyond Health Monitoring 23

11 Conclusion 24

1 The Multi-Agent Problem

The story begins the same way for nearly every team building on-device AI. You start with a single agent, perhaps an activity classifier, compiled as a native library and embedded in your application. It works well. Inference is fast, integration is straightforward, and the platform toolchain handles everything.

Then the product requirements grow. A sleep classifier is added, followed by a heart rate monitor and a stress analyser. Each new agent is compiled into the same binary. The application grows, but the architecture holds.

Then the requirements cross a boundary. The watch needs to contribute sensor data. A clinician dashboard needs access to patient trends. The regulatory team requires independent verification of safety-critical agents. The machine learning team wants to update the sleep classifier weekly without waiting for a full app store review cycle. And suddenly, the architecture that served you well at small scale is working against you at every turn.

This is not a hypothetical scenario. It is the lived experience of teams building multi-agent systems across health monitoring, industrial IoT, robotics, and smart infrastructure. The pattern is consistent: monolithic architectures impose costs that grow superlinearly with system complexity, and the transition to a more modular architecture becomes increasingly expensive the longer it is deferred.

This white paper provides a rigorous, quantitative comparison of two paradigms for on-device AI agents:

1. **App-native (monolithic)**: agents built within a single application using platform service mechanisms (Android Bound Services, iOS XPC, Linux D-Bus), which in practice produce monolithic architectures where all agents share a single process, a single deployment lifecycle, and a single failure domain. The limitations of app-native IPC force this coupling even when developers intend modularity.
2. **Agentix-native (microservice-based)**: each agent packaged as an independently deployable serverless microservice, termed a *mim* (micro intelligence module), with defined API contracts, running on a purpose-built agent operating environment (*mimOE*) that provides service discovery, lifecycle management, and continuum coordination. Model weights are stored in a shared registry, separate from the mim code, enabling independent update cycles and cross-agent model sharing. An Agentix-native system is a software system composed of collaborating mims that work within and across domains to drive outcomes, operating on principles of observe, respond, and learn.

We use continuous health monitoring as our primary reference application because it simultaneously exercises all seven comparison dimensions. However, the architectural conclusions generalize to any domain requiring multiple collaborating agents on user devices.

2 Device-First Continuum AI

Before comparing architectures, it is important to understand the computational paradigm within which both operate.

The Device-First Continuum AI (DFC-AI) architecture [6] positions computation along a contin-

uum from user devices through edge nodes to cloud infrastructure, building on the Hybrid Edge Cloud (HEC) paradigm [5]. Unlike cloud-centric architectures that treat devices as thin clients, or pure-edge approaches that require dedicated infrastructure, DFC-AI leverages the computational resources already present on billions of personal devices.

The key principles of DFC-AI are:

- **Device-first processing**: personal data is processed as close to the user as feasible, preserving privacy and enabling offline operation.
- **Calibrated self-awareness**: each agent knows the limits of its own competence and can escalate to more capable compute (a peer agent, an edge node, or the cloud) when a query exceeds its confidence threshold.
- **Lightweight, deterministic agents**: rather than deploying large language models on-device, DFC-AI favours small, purpose-built models (typically 2–12 MB) that deliver predictable latency and energy consumption.
- **Dynamic resource discovery**: agents can locate the most efficient and cost-effective compute resources available to them, whether on the local device, a nearby peer device, an edge gateway, or a cloud node, and make runtime decisions about where to execute based on current resource availability, latency constraints, and cost.
- **Continuum coordination**: agents can be deployed, migrated, and updated across any node in the device-to-edge-to-cloud continuum without recompilation.
- **Lifecycle management**: in multi-agent systems, lifecycle management is not a convenience but an architectural necessity. Without systematic startup sequencing, health monitoring, restart policies, and graceful degradation, multi-agent constellations become unmanageable. This is the operations, configuration, diagnostics, fault, and performance (OCD/FCAPS) problem that many teams building multi-agent systems discover too late: individual agents may work well in isolation, but managing their collective lifecycle across heterogeneous devices is a problem that grows combinatorially with agent count. DFC-AI treats lifecycle management as a first-class architectural primitive rather than an afterthought.
- **Zero trust security**: all inter-agent and inter-node communication is authenticated and authorised regardless of network location, with no implicit trust granted based on device proximity or network membership.

A note on terminology: the word "discovery" is overloaded in this domain. DFC-AI distinguishes between two types of discovery that *mimOE* provides. *Resource discovery* finds node-level capabilities: available compute capacity, hardware features such as GPU or NPU presence, memory headroom, and network connectivity. *Service discovery* finds which services exist and are reachable: their API contracts, endpoints, health status, and version metadata. *mimOE* does *not* provide semantic discovery, meaning it does not interpret what a service does or means. An agent can discover that a service exists at a given endpoint with a particular API contract, but must interrogate the service further to understand its functional capabilities. This distinction matters because readers familiar with AI agent frameworks may expect "capability-based discovery" to imply semantic understanding; in DFC-AI, it refers to structured metadata about service contracts.

The *mimOE* provides the runtime substrate for DFC-AI. It is a lightweight operating environment that runs on consumer hardware and enables microservice-based agents packaged as WebAssembly (WASM) modules. *mimOE* is installed as a library within a host application and exposes a set

of platform-independent APIs for agent lifecycle management, dynamic service discovery, dynamic resource discovery, zero trust security, and peer-to-peer communication across devices. It does not impose a centralised control plane or require a dedicated server; coordination is fully decentralised, with each device running its own *mimOE* instance and discovering peers directly. Agents register their capabilities with their local *mimOE* instance, which maintains a distributed service registry across all reachable nodes. When an agent requires compute resources beyond what is available locally, *mimOE*'s dynamic resource discovery identifies the most suitable node based on available capacity, latency, and cost. All communication between agents and nodes is secured through *mimOE*'s zero trust model, in which every request is authenticated and authorised regardless of whether the communicating agents are on the same device or across a network boundary. *mimOE* currently runs on iOS, Android, Linux, macOS, and Windows.

This white paper evaluates whether *mimOE* and the Agentix-native paradigm it enables are justified given the overhead they introduce, compared to the simpler app-native approach.

3 Architecture Definitions

3.1 Common Ground

Before defining each architecture, it is important to establish what they share. Both paradigms use identical deterministic, lightweight models for inference. Both execute on-device with identical privacy guarantees for local processing. Both achieve the same raw inference accuracy for any given model. The comparison is purely *architectural*: it concerns how agents are organized, deployed, updated, and coordinated, not what algorithms they run.

3.2 App-Native Architecture (Monolithic)

In the app-native paradigm, all inference models are compiled as native libraries and linked into a single application binary. The runtime environment is the device operating system itself. Inter-agent communication occurs through in-process function calls or shared memory regions, with negligible latency (below 0.01 ms). Updates require repackaging and redeploying the entire application binary through the platform app store, subjecting all agents to the store's review cycle regardless of which agent changed. All agents share a single process, a single deployment lifecycle, and a single failure domain.

As we demonstrate in Section 4, the limitations of platform service mechanisms (Android Bound Services, iOS XPC, Linux D-Bus, ROS 2 DDS) force this monolithic coupling even when developers intend modularity, because these mechanisms lack the fundamental primitives required for true agent independence.

3.3 Agentix-Native Architecture (Microservice-Based)

In the Agentix-native paradigm, each agent is packaged as an independently deployable serverless microservice, termed a *mim* (micro intelligence module), with defined API contracts. The *mimOE* provides the runtime substrate through a three-layer architecture:

- **Mim layer**: each mim contains the orchestration logic for a specific agent workflow, packaged as a lightweight WASM binary (typically under 500 kB). Mims operate as serverless processes: they spin up on invocation, execute their workflow, and release resources after a configurable keepalive

period. *mimOE* maintains a keepalive cache so that frequently invoked mims remain warm and avoid cold-start overhead.

- **Inference engine**: a shared runtime that executes model inference on behalf of mims. The inference engine manages model loading, GPU/NPU dispatch, and execution scheduling, providing a uniform inference interface regardless of the underlying hardware.
- **Model registry**: a shared repository of model weights with LRU caching. Models are stored independently of the mims that reference them, enabling multiple mims with different workflows to reuse the same model. When several agents share a common backbone (for example, a PPG feature extractor used by cardiac monitoring, anomaly detection, and stress analysis agents), the backbone is stored once in the registry and cached once in memory, eliminating duplicate storage and reducing memory consumption.

This three-layer separation creates two independent lifecycles with distinct caching behaviour:

- **Mim process cache** (short-lived, governed by `keepalive_ttl`): controls how long a mim instance remains in memory after its last invocation. The probability that a mim is cold (not cached) follows $p_{\text{cold,mim}} = e^{-f \cdot t_{\text{ka}}}$, where f is the invocation frequency and t_{ka} is the keepalive TTL.
- **Model registry cache** (longer-lived, governed by `model_cache_ttl`): controls how long model weights remain in the LRU cache. For shared models, the effective invocation frequency is the *combined* frequency of all agents referencing that model, keeping shared models warm even when individual agents invoke infrequently. The probability that a model is cold follows $p_{\text{cold,model}} = e^{-f_{\text{combined}} \cdot t_{\text{mc}}}$.

The expected per-invocation latency cost is:

$$t_{\text{invoke}} = (p_{\text{cold,mim}} \cdot t_{\text{cold}} + p_{\text{cold,model}} \cdot t_{\text{mload}} + t_{\text{inf}}) \cdot \alpha_w \tag{1}$$

where t_{cold} is the mim cold-start time, t_{mload} is the model load time, t_{inf} is the native inference latency, and α_w is the WASM overhead multiplier.

This architecture enables the following runtime behaviours:

- **Dynamic service discovery** allows agents to find and query peers by their API contracts and metadata rather than by hardcoded identity or address. Mims can declare their functions and expose their resources through standard protocols such as the Model Context Protocol (MCP), enabling structured interoperability across the agent ecosystem.
- **Dynamic resource discovery** allows agents to locate available compute resources across the continuum and select the most efficient execution target.
- **Lifecycle management** handles agent startup sequencing, health monitoring, restart policies, and graceful degradation across the agent constellation.
- **Continuum coordination** enables agent deployment and migration across any node in the device-to-edge-to-cloud continuum without recompilation.
- **Zero trust security** authenticates and authorises every inter-agent and inter-node request regardless of network location.

Inter-agent communication uses runtime service calls with latency of 1 ms to 5 ms per query. Individual mims can be updated independently through over-the-air (OTA) delivery without affecting

other agents or requiring app store review. Model weights can be updated on a separate, typically less frequent, cycle.

4 Why App-Native Mechanisms Force Monolithic Coupling

A natural question arises: rather than introducing a purpose-built agent operating environment, why not use the service mechanisms already provided by mobile and desktop operating systems? Android offers Bound Services and AIDL. iOS provides XPC Services and App Extensions. Linux systems can use D-Bus. Robotics platforms have ROS 2 with DDS-based communication.

Each of these mechanisms provides some form of inter-process communication. However, none of them provides the full set of primitives that multi-agent on-device AI systems require, which is why teams that start with app-native services end up building monolithic architectures regardless of their original intent. Table 1 summarizes the comparison across ten critical capabilities.

Table 1: Comparison of App-Native Service Mechanisms and *mimOE*

Capability	**Android Bound Svc**	**iOS XPC**	**Linux D-Bus**	**ROS 2 DDS**	***mimOE***
Dynamic resource discovery	–	–	–	–	✓
Dynamic service discovery	–	–	–	Partial	✓
Cross-platform portability	–	–	–	–	✓
Continuum routing	–	–	–	–	✓
Zero trust security	–	–	–	–	✓
Memory-safe isolation	–	–	–	–	✓
Independent OTA updates	–	–	–	–	✓
Lifecycle management	Partial	–	Partial	Partial	✓
Uniform API contracts	–	–	–	Partial	✓
Agent health monitoring	–	–	–	Partial	✓

The following subsections examine each mechanism in detail.

4.1 Android Bound Services and AIDL

Android Bound Services allow inter-process communication between application components. A client binds to a service by specifying an explicit Intent with a package name and component class. This is *name-based binding*: the caller must know at compile time exactly which package and service class it wants to reach.

This has several consequences for multi-agent systems:

No dynamic service discovery. An agent cannot ask “who provides cardiac anomaly detection?” It must instead specify “connect me to `com.health.app/.AnomalyService`.” If the anomaly detection agent is replaced by a better implementation from a different developer, every dependent agent must be updated with the new package and class name.

No dynamic resource discovery. Android Bound Services have no visibility into compute resources beyond the local device. An agent cannot determine whether a nearby edge gateway or peer device has available capacity for a computationally intensive inference task.

No memory-safe isolation. Services within the same application run in the same process by

default. A native code crash in one agent's shared library will crash its entire process. WASM sandboxing, by contrast, provides memory isolation by construction.

Unpredictable lifecycle management. The Android operating system can terminate background services at any time under memory pressure. Each agent pair requires defensive reconnection code independently.

No independent updates. Updating any agent requires pushing the entire APK through the Google Play Store.

No zero trust security between agents. There is no per-request authentication or authorisation between agents within the same application.

Single platform only. Android Bound Services work exclusively on Android.

4.2 iOS XPC Services and App Extensions

Apple's inter-process communication is the most restrictive of any major platform, and for multi-agent AI systems, the restrictions are architectural, not merely inconvenient.

Only Apple can build XPC servers. The most fundamental limitation is that on iOS, only Apple's own frameworks can create XPC servers. Third-party developers are limited to XPC *clients*. This means that no third party can create independently discoverable agent services on iOS using the platform's native IPC mechanism. The entire concept of a multi-agent system where agents register, discover, and query each other through XPC is architecturally impossible for third-party developers. This single constraint eliminates iOS XPC as a viable foundation for on-device multi-agent systems.

Severely constrained inter-app communication. iOS XPC Services are designed for communication between an application and its own extensions, not between independent applications. Cross-application communication is limited to URL schemes, share extensions, and App Groups with shared containers.

Strict resource limits. App Extensions operate under tight memory constraints and execution time limits.[1]

No background execution model. iOS severely throttles background execution. Agents that must perform continuous inference cannot rely on iOS background execution modes.

No service discovery of any kind. iOS provides no mechanism for one application to discover services offered by another application at runtime.

No independent updates. The entire application bundle must pass through Apple's App Review process.

4.3 Linux D-Bus

D-Bus is the standard inter-process communication mechanism on Linux desktop and embedded systems.

Name-based discovery only. Discovery is by well-known bus name, not by service contract or capability metadata.

[1] Apple does not publish fixed limits; reported values vary by extension type and iOS version. Developers report typical limits of 50–120 MB for memory and 30 seconds for execution in App Extensions.

Partial lifecycle management. D-Bus supports basic service activation but provides no health monitoring, automatic restart, or graceful degradation.

No cross-device or continuum support. D-Bus operates within a single machine.

Linux only. D-Bus does not run on iOS or Android.

4.4 ROS 2 and DDS

ROS 2 is the most architecturally comparable to *mimOE* among existing platforms. However, ROS 2 is primarily a prototyping and research environment and is not typically suitable for production deployment or product delivery.

Topic-based, not contract-based, discovery. ROS 2 nodes subscribe to named topics, which is more flexible than name-based lookup but still requires agents to know topic names in advance.

No dynamic resource discovery. Execution is determined by deployment configuration, not by runtime resource availability.

Heavy runtime footprint. A minimal ROS 2 installation typically exceeds 200 MB, an order of magnitude larger than *mimOE*'s 22 MB footprint.

No mobile OS support. ROS 2 does not run on iOS or Android.

No OTA agent update mechanism. Updates typically require rebuilding and redeploying the entire workspace.

No device-to-edge-to-cloud continuum awareness.

4.5 Summary: From App-Native Constraints to Agentix-Native Primitives

The comparison reveals five fundamental primitives that no app-native mechanism provides. Together, these primitives define what it means to be Agentix-native:

1. **Dynamic resource discovery**: the ability for an agent to locate the most efficient and cost-effective compute resources available across the continuum. No app-native mechanism provides this. An agent using Android Bound Services or iOS XPC has no visibility into compute resources beyond its own device.
2. **Dynamic service discovery**: the ability for an agent to find peers by their API contracts and metadata, rather than by their address or name. It is important to note that *mimOE* provides the discovery mechanism and exposes service metadata; semantic interpretation of what a service does remains the agent's responsibility.
3. **Continuum-transparent routing**: the ability for a query to reach the right agent regardless of whether it is on the same device, a nearby device, or an edge node.
4. **Zero trust security**: all inter-agent and inter-node communication authenticated and authorised regardless of network location.
5. **Memory-safe isolation**: the guarantee that one agent cannot corrupt another agent's memory, provided by WASM sandboxing with each module operating in its own linear memory space.

These are the architectural primitives that define an Agentix-native system. A development team that chooses app-native services will need to build these primitives themselves. The resulting accu-

mulated infrastructure converges on a purpose-built agent operating environment. The decision is whether to build this infrastructure from scratch or use *mimOE*.

5 Quantitative Analysis Framework

5.1 Parameterization from Published Benchmarks

All model parameters are derived from peer-reviewed publications and published benchmarks, ensuring full reproducibility. Table 2 summarizes the key parameters with their published ranges, including the serverless cache parameters introduced in this analysis.

Table 2: Simulation Parameters

Parameter	**Symbol**	**Low**	**Nom.**	**High**	**Unit**
Execution & Energy					
WASM vs. native overhead [11]	α_w	1.2	1.45	2.0	×
Mim cold start [12, 13]	t_{cold}	5	15	50	ms
Energy per inference [14, 15]	E_{inf}	10	25	50	mJ
OE per-inference overhead [20]	E_{oe}	0.5	2.0	5.0	mJ
IPC energy per call [21]	E_{ipc}	0.05	0.15	0.5	mJ
Smartphone battery	B_{ph}	15	17.5	20	Wh
Native inference latency [14, 17]	t_{inf}	5	15	40	ms
Communication					
IPC latency [22, 10]	t_{ipc}	1.0	2.5	5.0	ms
In-process call latency	t_{call}	0.001	0.005	0.01	ms
Memory					
OE memory overhead [13]	M_{oe}	15	22	30	MB
Per-mim process overhead [19]	M_{proc}	1	3	5	MB
Serverless Cache					
Mim keepalive TTL	t_{ka}	10	30	120	s
Model cache TTL	t_{mc}	60	300	600	s
Model load time	t_{mload}	5	20	80	ms
Update & Deployment					
Mim code size	S_{mim}	0.1	0.5	1.0	MB
Model update ratio	r_{mupd}	0.1	0.3	0.5	
Model dedup ratio	r_{dedup}	0.3	0.45	0.6	
App store review cycle	t_{rev}	1	3	7	days
Full app binary size	S_{app}	30	60	120	MB
Failure & Recovery					
Agent failure rate [23]	p_{fail}	0.001	0.005	0.02	/day
Cascade failure prob. [23]	p_{cas}	0.3	0.5	0.8	
Full app restart time [23]	$t_{\text{rest,M}}$	2	5	15	s
Mim restart time	$t_{\text{rest},\mu}$	0.05	0.1	0.5	s
Regulatory & Engineering					
Full system verification [24]	V_{full}	4	8	16	p-wk
Unit verification [24]	V_{unit}	0.5	1.5	3.0	p-wk
Platform effort multiplier [28]	E_{plat}	1.3	1.6	2.0	×/plat

5.2 Reference Workload: Health Monitoring

We define a constellation of twelve health monitoring agents spanning cardiac, respiratory, metabolic, sleep, activity, and behavioural domains. Table 3 lists each agent with its inference frequency, model characteristics, and shared model group. Agents sharing a backbone model (PPG, IMU, or HRV) benefit from deduplication in the model registry. The total workload is approximately 76,000 inferences per day across all twelve agents.

Table 3: Health Monitoring Agent Constellation

Agent	Freq. (Hz)	Model (MB)	Mim (MB)	RAM (MB)	Safety Critical	Model Group
Heart Rate Monitor	0.200	3.0	0.4	8	✓	PPG
Sleep Classifier	0.033	5.0	0.5	12		HRV
Stress/HRV Analyser	0.017	4.0	0.3	10		PPG
Activity Classifier	0.500	6.0	0.5	10		IMU
Anomaly Detector	0.033	8.0	0.5	15	✓	PPG
Nutrition Tracker	0.001	10.0	0.4	12		—
Medication Adherence	0.0003	2.0	0.2	5	✓	—
Preventive Coordinator	0.0001	12.0	0.5	20		—
Respiratory Monitor	0.100	4.0	0.3	9	✓	IMU
Skin Temp. Analyser	0.017	2.5	0.2	6		HRV
Blood Pressure Est.	0.003	7.0	0.4	14	✓	PPG
Mental Health Screener	0.0003	9.0	0.5	16		—
Total		**72.5**	**4.7**	**137**	**5 of 12**	

Three shared backbone models enable cross-agent deduplication in the model registry:

- **PPG Feature Extractor** (2.1 MB): shared by Heart Rate Monitor, Stress/HRV Analyser, Anomaly Detector, and Blood Pressure Estimator (4 agents).
- **IMU Motion Extractor** (1.4 MB): shared by Activity Classifier and Respiratory Monitor (2 agents).
- **HRV / Circadian Model** (1.8 MB): shared by Sleep Classifier and Skin Temperature Analyser (2 agents).

The remaining four agents (Nutrition Tracker, Medication Adherence, Preventive Coordinator, Mental Health Screener) use standalone models with no deduplication.

All agents run on the smartphone. Wearable devices serve as sensor sources streaming raw data (PPG, accelerometer, temperature) to the phone; they do not execute agent inference. Edge nodes serve as escalation targets for computationally intensive inference that exceeds on-device capacity.

5.3 Energy Model

For the monolithic architecture, daily energy consumption is:

$$E_{\text{mono}} = \sum_{i=1}^{N} f_i \cdot T \cdot E_{\text{inf}} \tag{2}$$

where f_i is the inference frequency of agent i, $T = 86\,400\,\text{s}$ is one day, and E_{inf} is the per-inference energy for native execution.

For the microservice architecture:

$$E_{\text{ms}} = \sum_{i=1}^{N} f_i \cdot T \cdot (\alpha_w \cdot E_{\text{inf}} + E_{\text{oe}} + q_i \cdot E_{\text{ipc}} + E_{\text{cold},i}) \tag{3}$$

where α_w is the WASM execution overhead multiplier, E_{oe} is the per-inference operating environment overhead, q_i is the number of inter-agent queries per inference for agent i, E_{ipc} is the energy per IPC call, and $E_{\text{cold},i} = p_{\text{cold,mim}} \cdot t_{\text{cold}} \cdot P_{\text{inf}} + p_{\text{cold,model}} \cdot t_{\text{mload}} \cdot P_{\text{inf}}$ is the expected cold-start energy per invocation, with P_{inf} being the instantaneous inference power draw.

5.4 Serverless Memory Model

The monolithic architecture keeps all agents resident at all times:

$$M_{\text{mono}} = \sum_{i=1}^{N} (S_{\text{model},i} + M_{\text{runtime},i}) \tag{4}$$

The microservice architecture uses a serverless model where expected memory reflects only the probabilistically warm components:

$$M_{\text{ms}} = M_{\text{oe}} + \sum_{i=1}^{N} p_{\text{warm,mim},i} \cdot (M_{\text{runtime},i} + M_{\text{proc}} + S_{\text{mim},i}) + \sum_{j \in \mathcal{M}} p_{\text{warm,model},j} \cdot S_{\text{model},j} \tag{5}$$

where $\mathcal{M}$ is the set of *unique* models (after deduplication), $p_{\text{warm,mim},i} = 1 - e^{-f_i \cdot t_{\text{ka}}}$, and $p_{\text{warm,model},j} = 1 - e^{-f_{\text{combined},j} \cdot t_{\text{mc}}}$.

5.5 Simulation Scenarios

We evaluate six scenarios, each exercising different subsets of the seven comparison dimensions:

Scenario 1 (Scaling): System behaviour as agent count increases from 3 to 12, measuring energy, memory, communication overhead, and coupling complexity.

Scenario 2 (Composition): A population of 1,000 users with heterogeneous agent requirements, comparing universal bundling against dynamic per-user provisioning with model deduplication.

Scenario 3 (Evolution): Twelve months of continuous operation with independent code and model update cycles, comparing cumulative update payloads, review delays, and regression risk.

Scenario 4 (Resilience): Monte Carlo fault injection (10,000 simulations, 30 days each), comparing failure propagation and recovery behaviour.

Scenario 5 (Continuum Mobility): Multi-platform deployment across smartphone and edge node, comparing engineering effort and agent migration capability.

Scenario 6 (Regulatory Change Control): Twelve months of code and model updates evaluated against IEC 62304 verification requirements, comparing change control scope.

6 Results: Seven Dimensions Compared

6.1 Dimension 1: Evolvability

Over twelve months of simulated operation, the monolithic architecture accumulated 2.34 GB of cumulative update payload compared to 0.067 GB for the microservice architecture, a 97% reduction (Figure 1a). For code-only mim updates (workflow logic changes without model retraining), the microservice payload was just 0.014 GB, a 99.4% reduction versus the monolithic approach. For a fleet of 100,000 devices, this translates to 240 TB versus 6.8 TB of cumulative bandwidth.

The difference arises from two architectural separations. First, monolithic updates require redeploying the full application binary (60 MB nominal) regardless of which agent changed, while the microservice architecture deploys only the changed mim (typically under 500 kB) or the updated model weights (2 MB to 12 MB). Second, code and model updates follow independent cycles: over twelve months, the simulation produced 34 code updates and only 6 model updates, reflecting the reality that model retraining is expensive and batched less frequently than code changes.

The monolithic architecture accumulated 120 days of cumulative app store review delay across all updates. The microservice architecture bypasses app store review entirely through direct OTA delivery, achieving sub-hour time-to-deploy.

The monolithic architecture experienced 11 regression events over twelve months compared to 1 for the microservice architecture (Figure 1b).

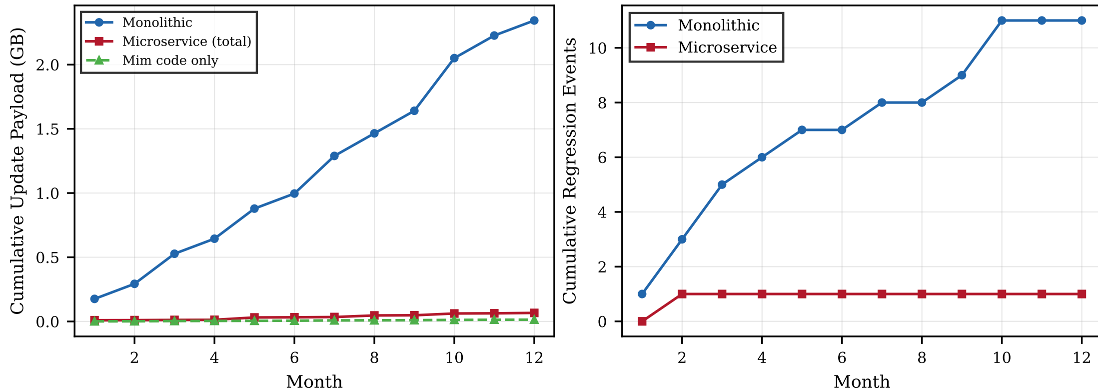

Figure 1: Twelve-month evolution simulation. (a) Cumulative update payload, showing monolithic (full binary redeployment), microservice total, and code-only mim updates. (b) Cumulative regression events caused by cross-agent interference during updates.

6.2 Dimension 2: Composability

We modelled 1,000 users with heterogeneous agent requirements drawn from empirical need probabilities. The average user required 5.5 of the 12 available agents.

Under the monolithic architecture, every user receives all twelve agents (72.5 MB of model storage), yielding 54% unused agent waste. Under the microservice architecture with model registry deduplication, each user receives only the mims and models they need, with mean storage of 29.8 MB, a 59% reduction (Figure 2).

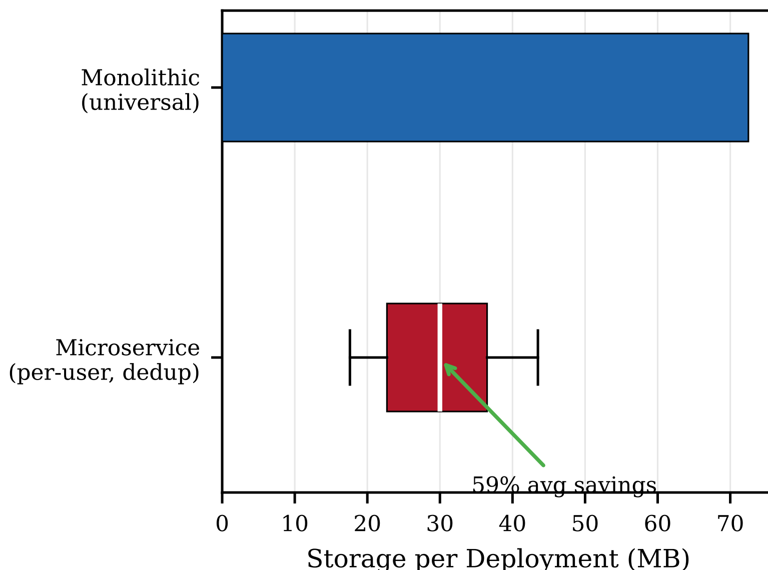

Figure 2: Per-deployment storage comparison. The monolithic architecture ships all twelve agents to every user regardless of need. The microservice architecture provisions only the mims and models each user requires, with model deduplication, yielding 59% average storage savings.

6.3 Dimension 3: Inter-Agent Reasoning

The inter-agent communication overhead introduced by the microservice architecture is real but bounded. At 8 agents, inter-agent queries through IPC account for 16.7% of the total per-inference latency for steady-state (warm) invocations, compared to 0.06% for in-process function calls in the monolithic architecture.

A typical health monitoring inference takes 15 ms natively. The microservice architecture adds 6.75 ms of WASM overhead and 2.5 ms to 5 ms per inter-agent query. For agents operating at intervals of seconds to minutes, total latencies of 22 ms to 27 ms are negligible relative to the monitoring interval. Low-frequency agents may occasionally incur additional cold-start latency when their mim or model must be loaded from storage, but the serverless cache architecture (Section 3.3) ensures that high-frequency agents, which dominate the inference workload, are almost always warm.

The architectural advantage becomes apparent as agent count grows (Figure 3). Monolithic coupling complexity scales quadratically: at 12 agents, there are 66 potential coupling edges. Microservice complexity scales linearly: 12 agents means 12 independent build-and-test units plus API contract verification.

6.4 Dimension 4: Regulatory Modularity

Five of the twelve health monitoring agents are safety-critical under IEC 62304 classification.

Over twelve months, the monolithic architecture accumulated 308 person-weeks of verification effort, compared to 89 person-weeks for the microservice architecture, a 71% reduction (Figure 4). The monolithic verification timeline totalled 96 weeks, versus 23.5 weeks for the microservice approach (76% reduction).

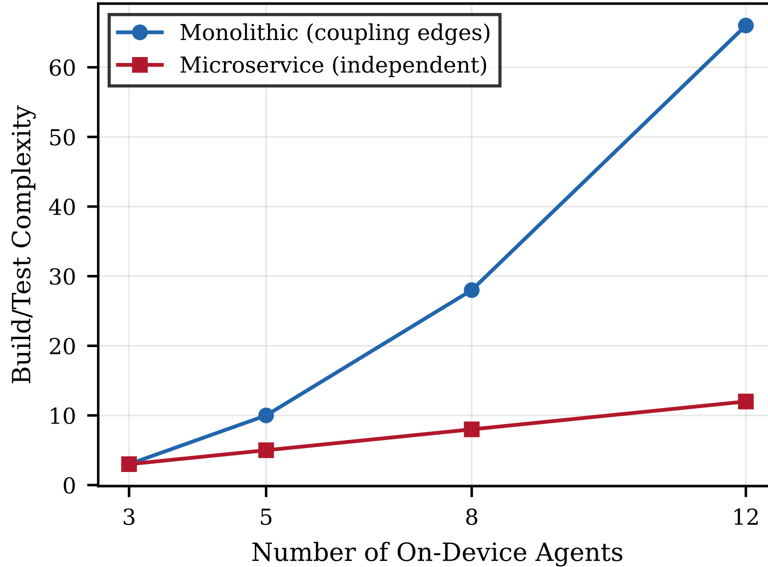

Figure 3: Build and test complexity as agent count grows. Monolithic coupling scales quadratically ($n(n-1)/2$ potential dependency edges) while microservice complexity scales linearly (n independent units).

The separation of code and model updates further reduces verification scope: code-only mim updates that do not change inference behaviour require lighter verification than model weight updates that alter prediction characteristics.

This maps directly to the FDA's predetermined change control plan framework and to IEC 62304's requirement for independence of software units at different safety classifications.

6.5 Dimension 5: Continuum Mobility

We modelled a deployment spanning smartphone and edge node, with wearable devices serving as sensor sources. The monolithic architecture required 35.7 person-months of total engineering effort. The microservice architecture required 9.7 person-months, a 73% reduction (Figure 5).

The microservice architecture enables live agent migration with latency of approximately 55 ms. A mim running on the smartphone can be migrated to an edge node without recompilation. The monolithic architecture has no migration capability.

6.6 Dimension 6: Operational Overhead

Energy: The microservice architecture incurs a consistent 58% energy overhead relative to native execution (Figure 6). At 12 agents, daily consumption is 3.1% of smartphone battery for native and 5.0% for microservice.

Memory: The serverless memory model produces a striking result: at 8 agents, the microservice architecture uses *less* expected memory than the monolithic architecture (99 MB versus 142 MB, a 30% *reduction*), and at 12 agents the reduction reaches 40% (125 MB versus 210 MB). This occurs because low-frequency agents consume zero memory between invocations in the serverless model but remain permanently resident in the monolithic architecture (Figure 7).

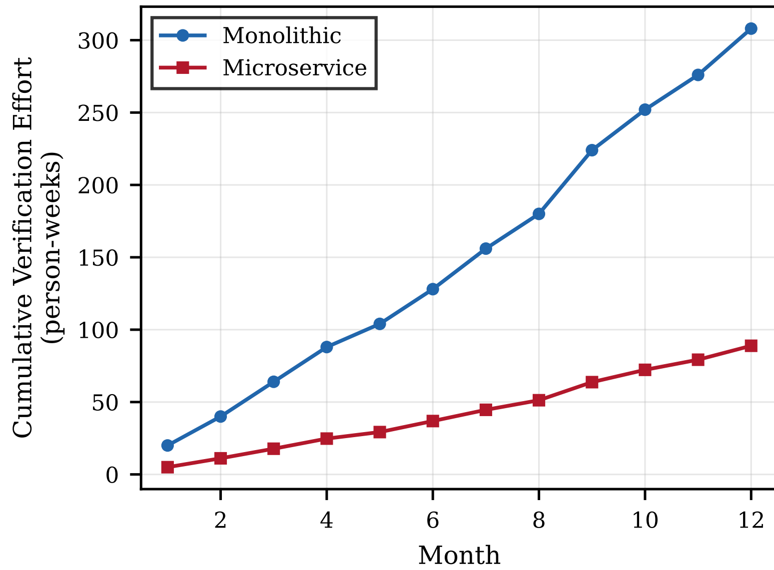

Figure 4: Cumulative regulatory verification effort over twelve months. Monolithic updates trigger full system re-verification; microservice updates require only unit-level verification plus API contract confirmation.

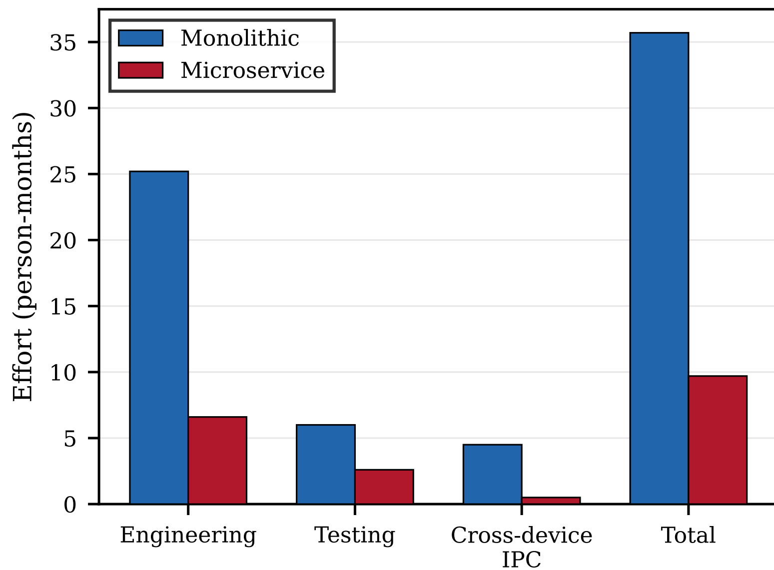

Figure 5: Engineering effort for multi-platform deployment (smartphone + edge node). The microservice architecture reduces total effort by 73% through platform-portable mims and the *mimOE* continuum fabric.

Peak memory (worst-case all agents warm simultaneously) shows a 25–32% overhead at 8–12 agents. However, this peak is rarely reached in practice because agents with diverse invocation frequencies are unlikely to all be warm at the same instant.

Cold-start behaviour: Figure 8 shows the cold-start probabilities for each agent. The key architectural insight is visible in mid-frequency agents: the Anomaly Detector (0.033 Hz) has a 61% mim cold-start probability but near-zero model cold-start probability, because its PPG backbone model is kept warm by the combined invocation frequency of all four PPG-group agents.

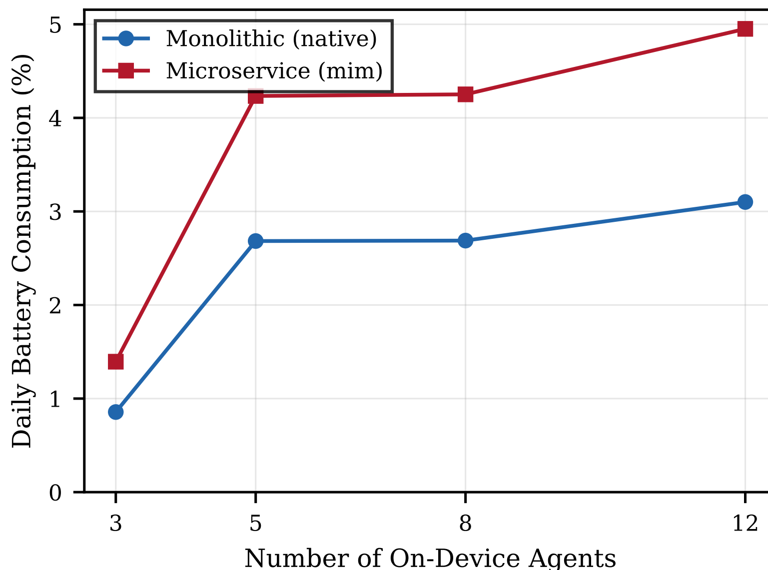

Figure 6: Daily smartphone battery consumption as agent count scales from 3 to 12. The microservice architecture incurs a consistent 58% energy overhead but remains at 5.0% of battery at 12 agents.

6.7 Dimension 7: System Resilience

Monte Carlo simulation (10,000 runs, 30 days, 12 agents) reveals a 7.8× blast radius reduction under the microservice architecture (Figure 9). In the monolithic architecture, a cascade failure crashes all agents, requiring a 5 s full application restart. In the microservice architecture, failure is confined to the affected mim, with a 100 ms restart, a 50× improvement in mean time to recovery.

Mean 30-day availability was 0.999997 for monolithic versus 0.9999999 for microservice. In the monolithic architecture, a crash in the nutrition tracker can take down the heart rate monitor. In the microservice architecture, agent failures are independent by construction.

7 Sensitivity Analysis

To confirm that these conclusions are not artefacts of particular parameter choices, we performed sensitivity analysis by varying each parameter across its full published range while holding others at nominal values.

Figure 10 shows the sensitivity of energy overhead. The WASM overhead multiplier dominates,

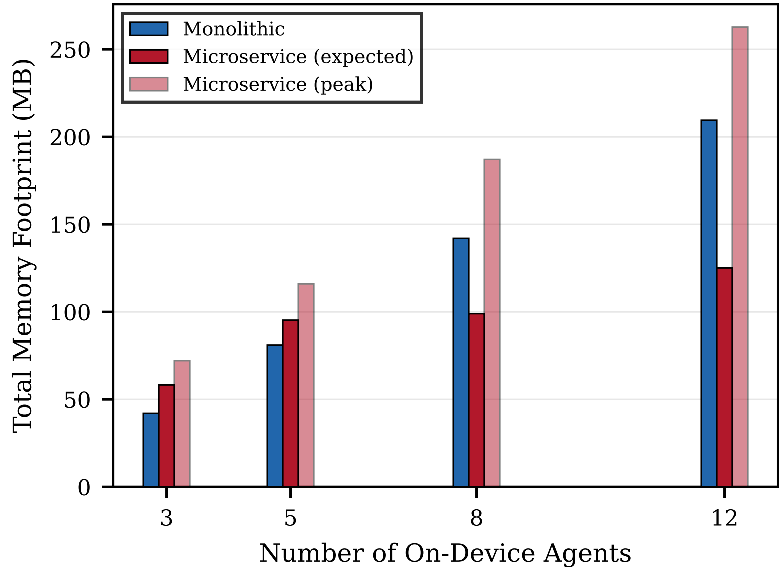

Figure 7: Total memory footprint. The serverless microservice model shows *lower* expected memory than monolithic at 8+ agents because low-frequency agents release resources between invocations. Peak memory (all agents warm) shows bounded overhead.

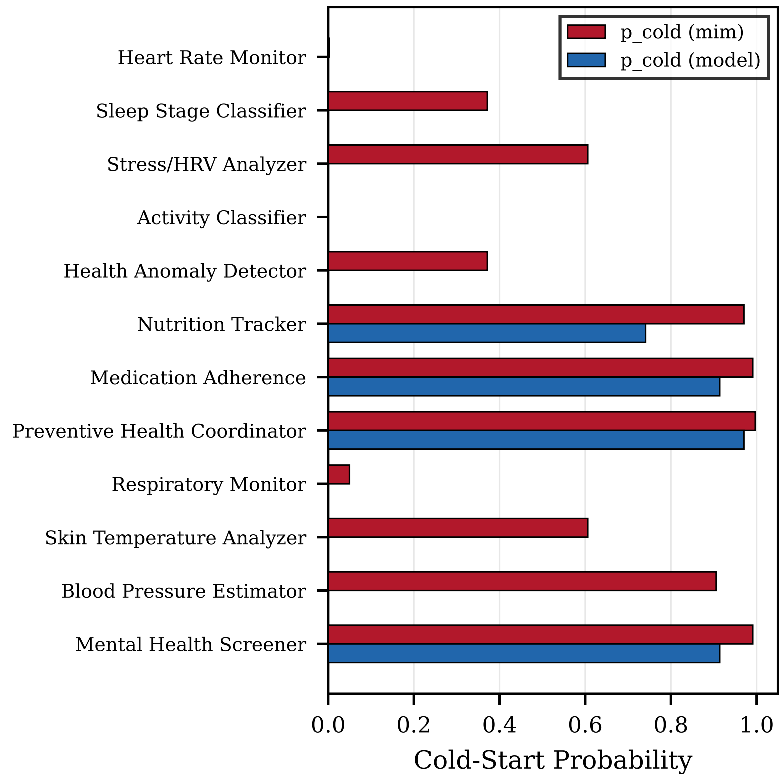

Figure 8: Cold-start probabilities per agent (keepalive TTL = 30 s, model cache TTL = 300 s). Mid-frequency agents like the Anomaly Detector show the key pattern: mim often cold but shared PPG model stays warm.

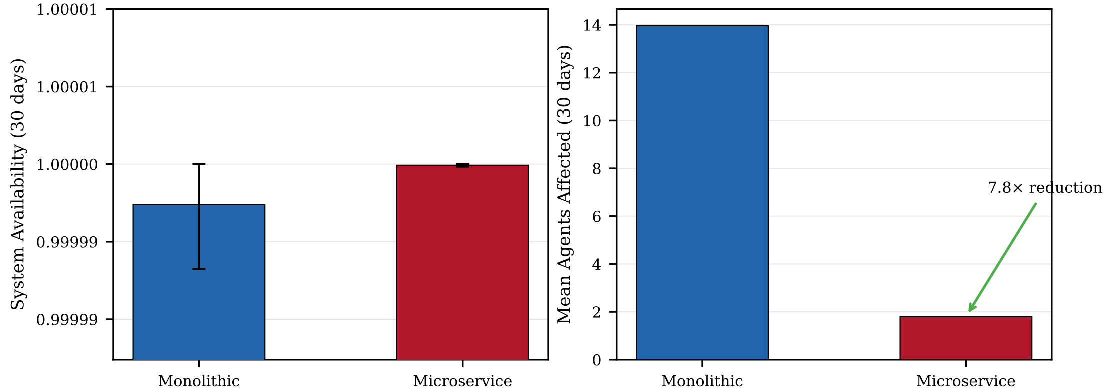

Figure 9: Monte Carlo resilience simulation (10,000 runs, 30 days). (a) System availability. (b) Blast radius: microservice architecture achieves a 7.8× reduction.

with an 80-percentage-point swing across its published range. Even at worst-case parameters, energy consumption remains below 6% of smartphone battery.

Figure 11 shows the sensitivity of the resilience benefit. Both restart times and cascade probability significantly influence the microservice advantage. Even under the most favourable monolithic assumptions, the resilience benefit remains above 7.5×.

Figure 12 shows the sensitivity of the memory overhead. The keepalive TTL is the dominant parameter. Across the full parameter range, the microservice architecture maintains a memory advantage at 8+ agents.

8 Consolidated Comparison

Table 4 consolidates the comparison across all seven dimensions for an 8-agent configuration.

9 Why the App-Native Assumption Fails in Practice

9.1 The Isolation Fallacy

The app-native monolithic architecture's efficiency advantage rests on an assumption that rarely holds in practice: that agents operate in complete isolation within a single application process on a single device with no external communication requirements. In reality, agents must communicate across applications on the same device, across devices owned by the same user, across devices owned by different parties (patient and clinician, drone and fleet controller, robot and supervisor), and across the device-to-edge-to-cloud continuum.

The moment an agent communicates beyond its own process, it requires service discovery, API contracts, and serialization: the core primitives of an Agentix-native architecture. As demonstrated in Section 4, app-native service mechanisms do not provide these primitives in a form suitable for multi-agent AI systems. The app-native approach therefore forces developers to build these primitives from scratch for each integration, progressively reinventing the Agentix-native operating environment at greater cumulative cost.

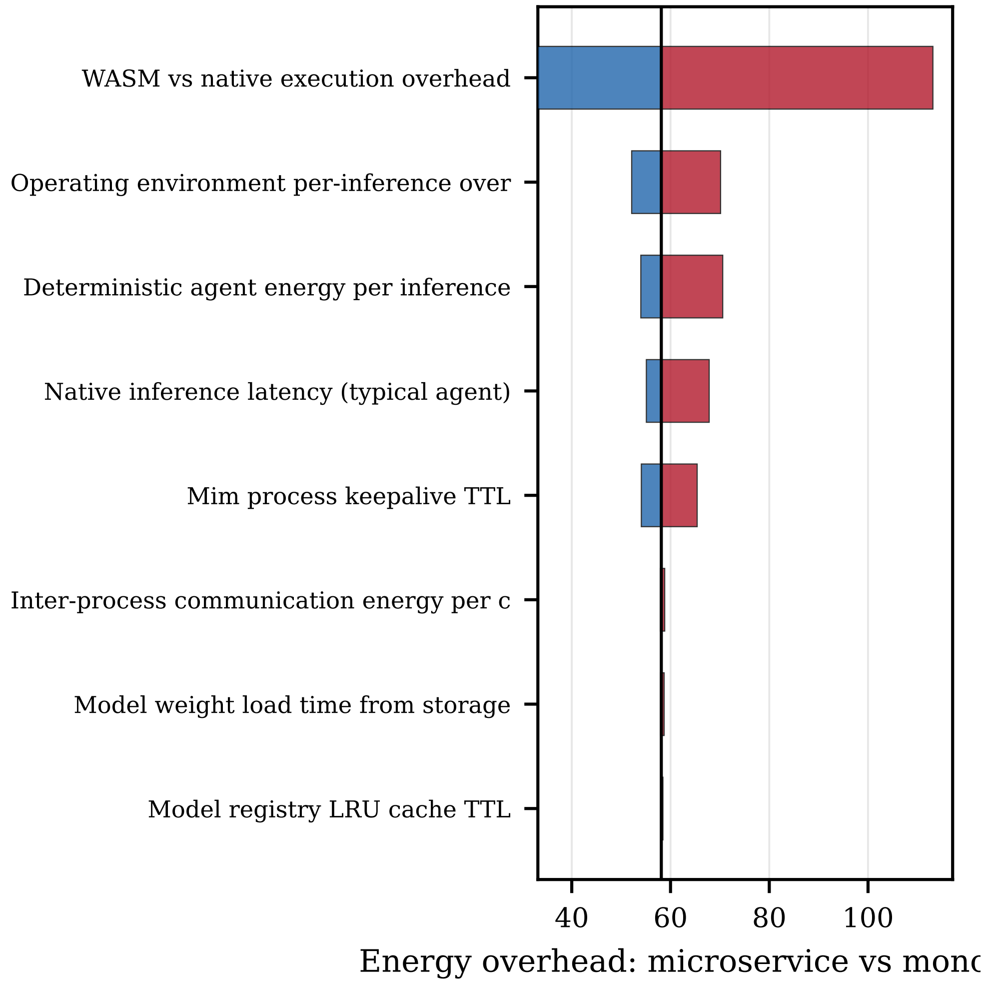

Figure 10: Tornado diagram for energy overhead sensitivity. The WASM overhead multiplier dominates (80-point swing).

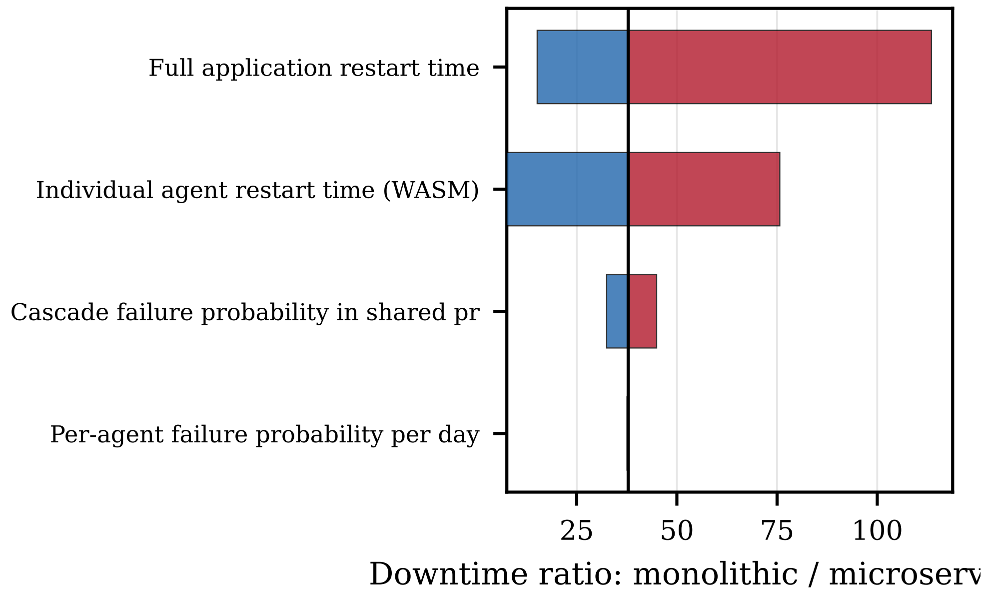

Figure 11: Tornado diagram for resilience benefit sensitivity. Both restart times and cascade probability significantly influence the microservice advantage.

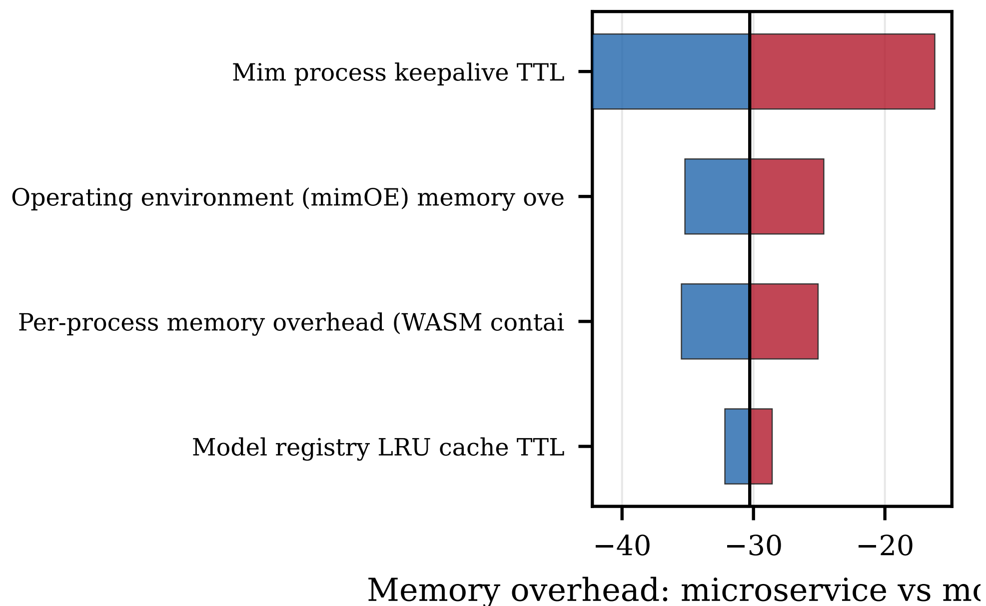

Figure 12: Tornado diagram for memory overhead sensitivity. The keepalive TTL dominates: shorter TTLs increase the serverless memory advantage.

Table 4: Architecture Comparison Summary: Seven Dimensions (8-Agent Configuration)

Dimension	Metric	App-Native (Monolithic)	Agentix-Native (Microservice)	Interpretation
Evolvability	12-month update payload	2.34 GB	0.067 GB	97% reduction
	Code-only payload	—	0.014 GB	99.4% reduction
	Regression events (12 mo.)	11	1	Near-eliminated
	Time-to-deploy per update	3 days	<1 hour	App store bypass
Composability	Storage per deployment	72.5 MB	29.8 MB	59% savings
	Unused agent waste	54%	0%	Dynamic provisioning
Inter-Agent Reasoning	Communication overhead	0.06%	16.7%	Bounded, negligible
	Coupling complexity (8 ag.)	28 edges	8 units	Quadratic vs. linear
Regulatory Modularity	12-month verification effort	308 p-wk	89 p-wk	71% reduction
Continuum Mobility	Multi-platform effort	35.7 p-mo	9.7 p-mo	73% reduction
	Agent migration	N/A	55 ms	Enabled by portability
Operational Overhead	Daily energy (8 ag., % batt)	2.69%	4.25%	58% overhead
	Memory expected (8 ag.)	142 MB	99 MB	30% *lower*
	Memory peak (8 ag.)	142 MB	187 MB	32% overhead
System Resilience	30-day availability	0.999997	0.9999999	Process isolation
	Blast radius	14.0 agents	1.8 agents	7.8× reduction

9.2 Architectural Discontinuity

The cost of app-native integration is not merely high; it is structurally discontinuous. A monolithic app-native architecture gives you a single set of primitives: function calls, shared memory, no serialization, no discovery. These primitives work perfectly within a single process on a single device. But the moment you cross any boundary, another process, another app, another device, another owner, those primitives stop working entirely. You do not get a degraded version of the same architecture. You get nothing. You must bolt on a completely different communication model from scratch: serialization, service contracts, discovery, security, and routing. Each boundary crossing is a discontinuity that forces a new engineering effort with no reuse from the previous one.

The Agentix-native architecture has no such discontinuity. An agent calling another agent uses the same mechanism, the same API contract, the same service discovery, and the same authenticated communication, whether the target is in the same process sandbox, on the same device, on a peer device across the room, or on an edge node in the cloud. The architecture is uniform across all boundaries. This uniformity is not a convenience; it is the reason that the cumulative integration effort remains flat as communication scope expands (Figure 13), while app-native integration effort grows with each new boundary crossed.

9.3 No Crossover Point

Unlike the cloud monolith-to-microservice transition, which exhibited a genuine crossover point where microservices became more efficient at sufficient scale, on-device agent systems face inter-agent communication requirements from their first deployment (Figure 13).

As communication boundaries expand from within a single app, to across apps on the same device, to across user devices, to across ownership boundaries, to across the device-to-edge-to-cloud continuum, the cumulative effort for ad-hoc integration grows to 20 person-months. The purpose-built operating environment achieves the same scope with 3 person-months, an 85% saving.

The question is not "when does the Agentix-native approach become worthwhile?" but "do you build the required infrastructure ad hoc on top of app-native services, or do you use a purpose-built agent operating environment?"

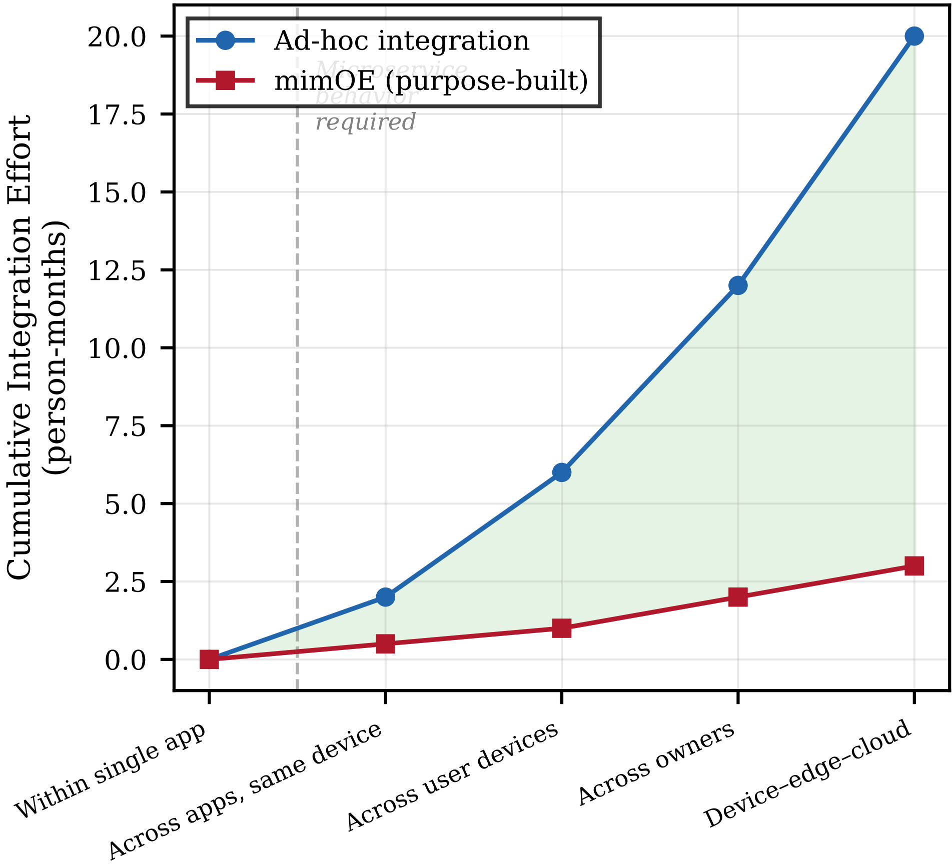

Figure 13: Cumulative integration effort as communication boundaries expand. Ad-hoc app-native integration accumulates 20 person-months; the Agentix-native operating environment achieves the same scope with 3 person-months (85% savings).

10 Beyond Health Monitoring

The seven comparison dimensions apply broadly across domains that require multiple collaborating agents on user devices. The specific parameter values differ, but the structural tradeoffs are invariant.

Industrial IoT. Factory floor monitoring systems deploy sensors and inference agents across heterogeneous controllers, gateways, and edge servers. Evolvability matters because models are retrained as process conditions change. Continuum mobility matters because agents must migrate between

floor-level controllers and plant-level edge servers. Regulatory modularity matters under IEC 61508 functional safety requirements.

Smart Infrastructure. Building management, energy grid monitoring, and transportation systems deploy agents across distributed nodes with varying computational capacity. Composability matters because different buildings need different agent constellations. Resilience matters because infrastructure monitoring must be fault-tolerant. Zero trust security is essential when agents communicate across administrative boundaries.

Personal AI Assistants. Multi-agent personal assistants that span phone, laptop, smart home devices, and wearables require exactly the continuum mobility and dynamic service discovery that *mimOE* provides.

Autonomous Systems and Robotics. Multi-robot systems and drone fleets require inter-agent communication across ownership boundaries, dynamic discovery of peers, and resilience to individual node failures. While ROS 2 addresses some of these needs in Linux-based research environments, it does not extend to the heterogeneous device ecosystem that increasingly participates in autonomous system architectures, nor does it provide the production-grade lifecycle management required for deployed systems.

In each domain, the pattern is consistent: the app-native monolithic approach works at small scale but imposes growing costs as the system requires more agents, more platforms, more frequent updates, and more inter-agent collaboration. The Agentix-native operating environment overhead is bounded and amortising; the costs it displaces grow with system complexity.

11 Conclusion

Through analytical modelling and simulation parameterised entirely by published benchmarks, we have demonstrated that the Agentix-native architecture, built on microservice-based mims with a serverless runtime model and shared model registry, fundamentally changes the tradeoff calculus for on-device multi-agent systems. The conventional wisdom that microservices trade efficiency for modularity holds only for energy (58% overhead, bounded below 5% of smartphone battery). For memory, the serverless model inverts the tradeoff entirely: expected memory usage is 30% *lower* than the app-native monolithic architecture at eight agents, because low-frequency agents consume zero resources between invocations rather than remaining permanently resident.

The architectural benefits compound across dimensions: 97% reduction in update payloads (99.4% for code-only updates) eliminates the bandwidth and review-cycle costs that make monolithic iteration expensive at fleet scale. The 59% storage savings from per-user provisioning with model deduplication respect the resource constraints of mobile devices. The 71% reduction in regulatory verification effort directly translates to faster iteration cycles for safety-critical applications under IEC 62304 and FDA predetermined change control plans. And the 7.8× blast radius reduction ensures that safety-critical agents are not collateral damage when a non-critical agent fails.

We have also shown that app-native service mechanisms do not provide the fundamental primitives that multi-agent on-device systems require: dynamic resource discovery, dynamic service discovery, continuum-transparent routing, zero trust security, and memory-safe isolation. These five primitives define what it means to be Agentix-native. The analysis of iOS XPC is particularly decisive: only Apple can build XPC servers, making the entire concept of third-party discoverable agent services architecturally impossible on iOS without a purpose-built agent operating environment.

The DFC-AI framework, with *mimOE* as its runtime substrate, provides this Agentix-native operating environment for the devices that billions of people already carry.

About mimik Technology. mimik Technology Inc. develops the *mimOE* operating environment and DFC-AI platform. Visit `developer.mimik.com` to explore the SDK and developer documentation.

References

[1] A. Rajkomar *et al.*, "Scalable and accurate deep learning with electronic health records," *npj Digital Medicine*, vol. 1, no. 18, 2018.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, 2016.

[3] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[4] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, 2017.

[5] S. M. Alamouti, F. Arjomandi, and M. Burger, "Hybrid Edge Cloud: A pragmatic approach for decentralized cloud computing," *IEEE Commun. Mag.*, vol. 60, no. 9, pp. 16–29, 2022.

[6] S. M. Alamouti, F. Arjomandi, M. Burger, and H. Gün, "Device First Continuum AI (DFC-AI): Realizing human-like AI," in *Proc. Future Technologies Conf. (FTC) 2025*, Lecture Notes in Networks and Systems, vol. 1676, Springer, 2026. arXiv:2501.14823.

[7] S. Newman, *Building Microservices.* O'Reilly Media, 2015.

[8] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, Springer, 2017, pp. 195–216.

[9] M. Villamizar *et al.*, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proc. IEEE ColCom*, 2015, pp. 583–590.

[10] A. Koschel, I. Astrova, and C. Kerschbaumer, "Microservices at the edge," in *Proc. IEEE IC2E*, 2021.

[11] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *Proc. USENIX ATC*, 2019, pp. 107–120.

[12] P. K. Gadepalli *et al.*, "Sledge: A serverless-first, lightweight wasm runtime for the edge," in *Proc. ACM/IEEE SEC*, 2020.

[13] L. Wang *et al.*, "WasmEdge: A lightweight, high-performance WebAssembly runtime," 2023.

[14] Google, "TensorFlow Lite benchmarks," 2023.

[15] Apple, "Core ML performance benchmarks," 2023.

[16] ONNX Runtime Contributors, "ONNX Runtime," 2023.

[17] X. Xu *et al.*, “MNN-AECS: Adaptive and efficient on-device neural network inference,” 2024.

[18] Y. Xue *et al.*, “PowerInfer-2: Fast large language model serving on smartphones,” 2024.

[19] A. Haas *et al.*, “Bringing the web up to speed with WebAssembly,” in *Proc. ACM PLDI*, 2017, pp. 185–200.

[20] Y. Zhu, E. Witchel, and J. Zheng, “Understanding the energy consumption of gRPC workloads,” in *Proc. IEEE IISWC*, 2022.

[21] A. Pathak, Y. C. Hu, and M. Zhang, “Where is the energy spent inside my app?,” in *Proc. ACM EuroSys*, 2012, pp. 29–42.

[22] Android Open Source Project, “Binder IPC mechanism,” 2023.

[23] Google, “Android Vitals: Technical quality benchmarks,” 2023.

[24] IEC 62304:2006+AMD1:2015, “Medical device software: Software life cycle processes,” 2015.

[25] IEC 61508, “Functional safety of E/E/PE safety-related systems,” 2010.

[26] U.S. FDA, “Software as a Medical Device (SaMD): Clinical evaluation,” 2017.

[27] U.S. FDA, “Predetermined change control plan for AI/ML-enabled device software,” Draft Guidance, 2023.

[28] A. Biørn-Hansen *et al.*, “Performance overhead in cross-platform mobile development,” *Empirical Softw. Eng.*, vol. 25, pp. 2997–3040, 2020.

[29] S. Laskaridis *et al.*, “SPINN: Synergistic progressive inference of neural networks,” in *Proc. ACM MobiCom*, 2020.

[30] A. Esteva *et al.*, “A guide to deep learning in healthcare,” *Nature Medicine*, vol. 25, pp. 24–29, 2019.