IT Project Guidance

Evaluating the Use of Microservice Architecture

Version: 0.3

Purpose

This document provides strategic guidance for organisations evaluating the suitability of microservice architecture in enterprise systems. It is intended to support architects, programme leads, business analysts, and procurement advisors in understanding the operational, organisational, and architectural trade-offs involved. The objective is not to dismiss microservices, but to support informed, context-sensitive decisions that favour architectural clarity, service continuity, and long-term maintainability. This guidance challenges the widespread perception that microservices are the default modern approach, and instead presents a balanced assessment of when and why microservices may be appropriate—and when they introduce unnecessary complexity.

Synopsis

Microservices are often promoted as a pathway to scalable, resilient, and modular digital services. Their appeal lies in the promise of faster release cycles, autonomous deployment, and parallel development streams. These benefits are real but context-dependent. They emerge primarily in hyperscale commercial platforms, where global reach, product diversity, and multi-team autonomy are core operational requirements. In contrast, most enterprises—particularly those serving a single geography, domain, or function—do not share these operational characteristics.

When adopted without these supporting conditions, microservices can result in a level of architectural and operational complexity that is out of proportion to the problems they are solving. In practice, they can obscure ownership, degrade reliability, and increase development and support overheads. This is particularly true in enterprises where systems are maintained by small or rotating teams, with limited delivery autonomy or infrastructure expertise. For these environments, a modular monolith—with clean internal boundaries, plugin capability, and well-structured, queryable interfaces—offers a more practical, transparent, and sustainable architecture.

Contents

Background

Microservice architecture was developed to meet the operational demands of globally scaled commercial enterprises. Organisations like Amazon and Netflix needed to release and manage components of large platforms without creating bottlenecks. Microservices provided a model that allowed each system function to evolve independently, governed by autonomous teams. This enabled parallel development at scale, supported continuous delivery, and protected the stability of the overall platform by isolating change.

However, this approach is tightly bound to the operating models of hyperscale digital businesses. These businesses have thousands of engineers working on multiple domains and are organised into dozens of product teams with full-stack ownership. They require deep decoupling and frequent deployment. Their engineering models are supported by extensive investment in cloud-native infrastructure, observability pipelines, and internal platform engineering teams. In short, microservices were built for a scale of complexity that few organisations face.

This architecture was further popularised by cloud vendors—particularly AWS—whose business model thrives on infrastructure proliferation. Fragmented systems that use distributed messaging, ephemeral compute, externalised logging, and infrastructure-managed orchestration increase consumption of the cloud services that they are one of the few to provide.

However, AWS is not a design authority; it is an infrastructure sales platform. Its guidance naturally reflects patterns that maximise the use of its offerings. This does not make those patterns wrong, but it makes them commercially motivated. Any architectural recommendation from a vendor must be interpreted through the lens of who benefits.

Most enterprises—including those in the public sector—do not operate at this scale. They typically serve a local or national user base, operate within a single domain, and focus on delivering a coherent, stable service. They often have fixed funding, fixed operating hours, and limited engineering capacity. While they may aspire to modular growth, this rarely eventuates, therefore almost never justifies the architectural and organisational overheads of a distributed system. Plug-in extensibility and clear domain modularity suitable to their needs with far less deployment capacity requirements can usually be delivered within a monolithic codebase using well-established practices.

Public sector organisations face these constraints in amplified form. They are almost never in a position to run independent delivery teams across domains. Their systems are maintained under central governance, with tightly coordinated planning and limited autonomy. In such cases, the rationale for microservices does not exist. The problem is not one of enabling parallel change—it is one of ensuring sustainable delivery within resource and skill constraints.

Current Context

Enterprise systems tend to follow long-lived planning cycles, stable funding profiles, and centrally managed release practices. Change is often driven by legislative updates, customer feedback, or procurement planning, not by the demands of product competition or global service reach. The systems in question are typically constrained in scope and run by small to medium-sized teams. In such environments, alignment across development streams is more important than independent deployment. The idea that different teams need to release different parts of a system on different cadences is largely theoretical.

In practice, enterprise systems are most concerned with clear ownership, reliable integration, and predictable cost. Microservices, if introduced without clear justification, complicate all three. Services need to be maintained, versioned, orchestrated, secured, and monitored—individually. The service boundaries must remain stable across time or incur integration drift. Logging must be centralised or duplicated. Message queues and asynchronous flows introduce nondeterminism. The operational burden increases with each service added, and dependency management becomes its own discipline.

A well-structured monolith offers a simpler path. It supports the same principles of separation of concerns and modular design within a single deployment unit. Features can be isolated by domain, exposed through service APIs, and even developed by distinct teams. Extensibility can be achieved through plugin mechanisms or modular configuration. Interoperability can be supported through queryable APIs. The key difference is that the deployment and runtime orchestration remain unified, reducing the number of moving parts and the risks they introduce.

In sum, most enterprises—including nearly all public agencies—are not operating under the delivery pressures or team scale that justify the complexity of microservices. Their systems benefit from clarity, not fragmentation. This document continues with an examination of the real risks, issues, and decision-making criteria involved in choosing between microservice and modular monolith architectures.

Reasons for Caution

The core argument in favour of microservices—the independent deployment and evolution of system components—depends on the presence of organisational structures capable of sustaining that independence. Without mature DevOps practices, full team ownership of services, dedicated infrastructure support, and rigorous contract management between components, the system devolves into complexity without coordination. Services become dependent in ways that were not designed for. Teams begin to share data models or infrastructure, negating the very isolation microservices intend to create.

The need to fragment functionality across multiple running services creates overheads at every layer. Developers must understand distributed systems even for basic changes. Testing becomes asynchronous and environment-sensitive. Deployment is no longer about application release but becomes a choreography of services. Operationally, monitoring and support must aggregate logs and performance signals from multiple sources. Errors must be traced across networks rather than stack traces. The cost of diagnosing faults, reconciling data, and maintaining availability increases proportionally to the number of services involved.

Crucially, most enterprise organisations are not structured to reap the benefits of this design. There is typically a single IT team or a small set of delivery contractors. There is no genuine autonomy or funding separation between groups. Releases are centrally planned. Even when development is outsourced to an external vendor or separated into a standalone team, the system ultimately returns to the core enterprise team for ongoing operation. This business-as-usual phase often lasts ten times longer than the development phase and defines the true cost, risk, and maintainability of the system. If there is no practical separation of accountability, then logically separating deployments only obscures responsibility. In effect, the illusion of independence becomes a liability.

Microservices also create invisible dependencies on cloud infrastructure. Systems become reliant on vendor-specific orchestration tools, message brokers, storage conventions, and security configurations. This increases cost sensitivity and complicates exit planning. Vendor features become part of the architecture rather than the environment. In high-control sectors such as public services, where policy, data ownership, and operational visibility are paramount, this can be particularly problematic.

As a design principle, microservices attempt to solve a deployment problem. When that problem does not exist—when delivery is already centralised—the introduction of this pattern adds technical debt instead of solving architectural bottlenecks.

Issues

Observed In practice, organisations that adopt microservices without a supporting environment often experience predictable patterns of difficulty. The most immediate issue is reduced visibility. Because system logic is scattered across multiple services, no single team or individual can easily trace business flow from start to finish. This can severely impact incident response and post-deployment verification. Understanding the full system requires distributed tracing tools, architectural discipline, and often a deep familiarity with inter-service protocols.

Second, delivery cadence does not improve. Contrary to expectations, services do not evolve independently. In the absence of autonomous teams with true product responsibility, microservices require tightly coordinated releases to prevent breaking changes across boundaries. This reintroduces the bottlenecks that microservices were designed to avoid, but without the cohesion and simplicity of a unified codebase.

Third, operational costs escalate. Every service must be deployed, monitored, and secured individually. Load testing, performance monitoring, and availability planning must account for network latencies, timeouts, retries, and cascading failures. This increases cloud infrastructure consumption, stretches platform teams thin, and multiplies the work required to maintain reliability.

Fourth, architectural drift becomes common. Without strong governance, services evolve in inconsistent ways. Some may become overly dependent on others, breaking the principle of autonomy. Others may duplicate data or logic unnecessarily, leading to divergence, confusion, and maintenance burden.

Finally, support and onboarding are hindered. New team members must grasp not only the business logic but also the choreography of distributed communication. Debugging requires knowledge of asynchronous workflows, queues, service meshes, and observability tooling. The steep learning curve erodes productivity and increases reliance on a shrinking number of specialists.

These outcomes are not speculative—they are widely reported in post-mortems and conference retrospectives across industries. They are the reason that many organisations, after experimenting with microservices, return to simpler, monolithic designs or hybrid models that preserve modularity without distributed deployment.

Risks

The adoption of microservices in enterprise settings introduces a broad range of architectural, operational, and strategic risks—particularly when implemented without matching organisational maturity or delivery scale.

The first major risk is misalignment between architecture and capability. Microservices rely on a high-functioning DevOps culture, advanced monitoring infrastructure, and the ability to manage distributed systems. When these capabilities are underdeveloped or absent, systems built on microservices become brittle. Partial adoption leads to fragmented architectures that lack the robustness of both monolithic and distributed models. Teams are forced to contend with the burden of microservices without the supporting tools, skills, or processes needed to manage them effectively.

A second risk is uncontrolled complexity. As services multiply, so do the interfaces between them. Each integration becomes a potential point of failure. Each additional service adds to the testing surface, deployment complexity, and coordination overhead. Even routine updates can trigger cascades of dependency checks and reconfiguration. Instead of making the system more modular, fragmentation can obscure where responsibilities lie, who owns which function, and how errors propagate.

The third risk lies in service coupling through shared dependencies. While microservices are intended to isolate responsibilities, they often share authentication services, data sources, event buses, or configuration stores. These shared dependencies act as choke points. A failure in one component can ripple across systems. Worse, tight coupling may emerge unintentionally, as teams adopt similar solutions to solve the same problems. Once introduced, reversing coupling is difficult and time-consuming.

There is also a strategic risk in cloud platform dependence. Microservice implementations often lean heavily on proprietary infrastructure: serverless functions, messaging layers, and monitoring stacks that are unique to a particular vendor. These tools are rarely interchangeable. What begins as a neutral deployment choice can quickly become a platform commitment with limited exit options. This form of soft lock-in increases costs over time and restricts architectural freedom.

Vendor failure modes add to this risk. Unlike with monolithic systems, where services are contained and observable as a whole, distributed systems rely on external guarantees. If an orchestration service stalls, a queue malfunctions, or a regional availability zone fails, the distributed application becomes unrecoverable without deep infrastructure knowledge. This risk is compounded when knowledge of individual services is siloed across multiple team members or contractors.

Finally, there is a risk of organisational disillusionment. Teams may embrace microservices for their promise of agility, but discover that delivery slows, bugs become harder to trace, and the overhead of managing infrastructure becomes the dominant cost. If services must still be coordinated, tested together, and released in step, then the rationale for splitting them is undermined. The return on architectural investment becomes negative.

In combination, these risks make microservices a high-stakes choice. They may offer advantages in particular enterprise conditions, such as federated delivery teams or high-scale public platforms. But when used without clear justification, they pose material risks to delivery certainty, operational transparency, architectural maintainability, and strategic independence.

Platform Architecture Considerations

A critical but often overlooked decision in system design is whether the enterprise will build a custom system to express its strategic intent, or adopt a pre-existing platform to align with external patterns. This is a foundational architectural choice—between expressing differentiation or accepting constraint.

Custom development allows a system to directly embody the organisation’s mission, workflow, and values. It offers the freedom to innovate, to shape capabilities that reflect unique strategies, and to evolve independently. But it carries significant delivery risk. It requires architectural maturity, a capable and cohesive team, and a sustained commitment to maintain and evolve the system responsibly. Organisations that lack these competencies often find themselves unable to deliver or sustain the outcomes they initially sought.

Platform-based architecture takes a different path. By adopting a commercial or community-developed platform, organisations outsource a portion of their flexibility in exchange for operational stability, scalability, and managed support. But this alignment comes at a cost: the platform's roadmap is not their own. Strategic intentions must be negotiated with vendor constraints. Capabilities may be gated by licensing, delayed by upstream decisions, or implemented in ways that reflect someone else's priorities.

This is not just a build-versus-buy question—it is a question of intent ownership. Do we wish to steer the capability, or to be passengers within a larger framework? The former demands venture and courage. The latter offers safety and resilience—but it will inevitably shape the way the organisation works, often in subtle, accumulating ways.

For many organisations, especially those in the public sector, the challenge is structural rather than optional. A government agency, for example, is not simply a service provider—it acts as the custodian of public intent. It represents the lived experiences, legal obligations, and national priorities of a country. In such contexts, strategic divergence from platform assumptions is not merely a technical preference—it is often a necessary expression of public responsibility.. The question becomes how to balance this with the realities of limited delivery capability.

The intent must be clear: to deliver public or enterprise services that express the organisation's distinct purpose, responsibilities, and identity. Architecture is not just about technology—it is about enabling that intent through the systems we choose to build, adopt, or govern. That intent must be followed by a deliberate decision to engage in a mission of architectural self-determination, recognising that while capability gaps may exist today, they do not excuse the permanent outsourcing of strategic control.

A pragmatic pathway can then be pursued: to lean on products and commercial platforms to reduce early complexity, engage delivery partners when necessary, and progressively develop internal capability to define and deliver the outcomes that reflect the organisation's own values. This transitional reliance should be seen not as a surrender of strategy, but as a scaffold for capability. The goal is to reach a position where the organisation can gradually assert its architectural intent—first in defining service boundaries, then in shaping service behaviours, and ultimately in owning their evolution. This approach does not reject platforms outright—it recognises their utility as scaffolding, not foundations. The end goal must remain the same: to regain control of architecture where it matters most. to reduce early complexity, engage delivery partners when necessary, and progressively develop internal capability to define and deliver the outcomes that reflect the organisation's own values. This approach does not reject platforms outright—it recognises their utility as scaffolding, not foundations. The end goal must remain the same: to regain control of architecture where it matters most.

Another important consideration is whether a platform is being adopted to support internal operations or to deliver external-facing services. Platforms used to support internal users—such as HR systems, finance tools, or document management solutions—may be tolerable trade-offs. The alignment with vendor priorities can be managed, as the consequences are largely contained to internal efficiency and workflow.

But platforms used to deliver public-facing services introduce far greater risk. In these cases, the system becomes the face of the organisation. Lock-in is no longer just a licensing concern—it becomes a structural constraint that shapes how the organisation serves its clients, expresses its values, and responds to change.

For government agencies, this risk becomes critical. Public institutions do not merely represent a brand—they embody a nation's legal obligations, social commitments, and democratic intent. When their external services are defined by the limits of a vendor platform, the consequence is not only operational. It is a constraint on sovereignty, policy agility, and public accountability.

Some organisations will choose a hybrid model: a core custom system augmented by platform tools, plugins, or commercial services. This can offer balance if governed carefully. But the risk remains that platform constraints eventually dictate more than intended. The decision should be deliberate, not accidental. It must be made with full awareness that architecture is not neutral—it encodes both the freedoms an organisation retains and the dependencies it accepts.

Recommendation

For most enterprise systems, particularly those delivering a single integrated service or operating under centrally governed constraints, a monolithic architecture remains not only sufficient, but optimal. In these environments, the overheads introduced by distributed systems offer negligible operational advantage and create disproportionate cost, risk, and technical complexity. A well-structured monolith offers a maintainable, transparent, and efficient foundation for service delivery.

Where internal modularity is needed, a modern modular monolith—or modulith—should be the first architectural evolution. This pattern enforces domain boundaries, enables clear separation of responsibilities, and allows internal plugin development without fragmenting the runtime or deployment model. It can be incrementally developed, tested cohesively, and deployed reliably by a single team.

Microservices should only be considered when a system is intended to operate as an extensible platform with third-party developers building and deploying independent components. Even then, the need for true runtime independence and external hosting must be proven. In such cases, it may be appropriate to deploy a bounded set of services specifically supporting the plugin layer, while maintaining the core application as a monolith. This hybrid model limits complexity to where it is operationally justified.

Platform adoption must be held to the same standard. Platforms used internally—such as for HR, finance, or operational tooling—may be acceptable trade-offs, offering efficiency without unduly compromising strategic direction. But platforms used to deliver public-facing or externally-consumed services carry far greater risk. In such cases, the organisation’s interface to its customers or citizens is mediated by vendor constraints. The result is not just technical lock-in—it is strategic and representational lock-in. For government agencies, the stakes are even higher. They embody national identity, legal obligations, and democratic intent. The architecture that delivers their services must remain under their control.

In nearly all other circumstances, the use of microservices or externally-governed platforms is architectural overreach. The theoretical gains they offer are often dwarfed by the complexity they introduce and the control they forfeit. Architects must weigh not just what is possible, but what is sustainable—and for whom.

The guiding principle should be this: if the system is a single service, build a monolith. If it requires internal modularity, evolve toward a modulith. If it exposes extensibility to external parties, consider selective microservice boundaries only where justified. If external functionality is required, it must not come at the cost of strategic intent. Simplicity, clarity, and long-term operability must remain the default.

Conclusion

Microservices are a deployment strategy, not an architectural necessity. Their value depends on the delivery environment, team structure, and long-term ownership model. In both enterprise and public systems, architecture must be a reflection of intent, capability, and accountability—not trend.

The push to adopt microservices, or externally governed platforms, must be tested against the specific operational realities and strategic responsibilities of the organisation. Public institutions in particular carry a civic burden—they represent the nation’s ability to define and deliver public value. When that capacity is outsourced to external architectures, the cost is not only financial but constitutional. Platform and service decisions shape what a country can do for its people.

Systems should be designed for transparency, stability, and maintainability. These goals are best achieved through clear modularity within a unified deployment model—typically a monolith or a modular monolith. Platforms may be used as scaffolding while capability is built, but not as substitutes for agency.

The use of microservices or platforms should be driven by structural necessity, not fashion. Architects must critically assess whether the complexity or dependency introduced is justified by the context. In most government and enterprise settings, the answer is a principled no.

A forthcoming appendix will provide implementation patterns and alternatives for modular monoliths that meet extensibility and domain separation needs without relying on service fragmentation.

In all cases, whether adopting services or platforms, the architecture must serve the public mission—not displace it.

Appendices

Appendix A - Document Information

Authors & Collaborators

- Sky Sigal, Solution Architect

Versions

0.1 Initial Draft

0.2 Reformat

0.3 Added Platforms

Images

No table of figures entries found.

Tables

No table of figures entries found.

References

There are no sources in the current document.

Review Distribution

The document was distributed for review as below:

Identity	Notes
Russell Campbell, Project Manager	
Carl Klitscher, Solution Architect	
Ken Matheson, Enterprise Architect	
Amy Orr, Enterprise Data Architect	
Vincent Wierdsma, Lead Developer	

Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

Structure

Where possible, the document structure is guided by either ISO-* standards or best practice.

Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

Acronyms

API : Application Programming Interface.

GUI: Graphical User Interface. A form of UI.

ICT: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

IT : acronym for Information, using Technology to automate and facilitate its management.

UI : User Interface. Contrast with API.

Terms

Refer to the project’s Glossary.

Application Programming Interface : an Interface provided for other systems to invoke (as opposed to User Interfaces).

User : a human user of a system via its UIs.

User Interface : a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces (GUI) or Text/Console User Interfaces (TUI).