

From shared libraries to sidecars: A layered approach to cross-cutting concerns in microservices

Ganesh Kumar Gangannagari *

Independent Researcher.

World Journal of Advanced Engineering Technology and Sciences, 2026, 20(01), 156–162

Publication history: Received on 31 May 2026; revised on 11 July 2026; accepted on 13 July 2026

Article DOI: <https://doi.org/10.30574/wjaets.2026.20.1.0354>

Abstract

Enterprise systems built as microservices decompose large applications into small, independently deployable units, and this decomposition exposes a recurring problem: certain capabilities, such as logging, authentication, tracing, and error handling, cut across nearly every service rather than belonging to any single one. Left unmanaged, this duplication multiplies maintenance effort and slows delivery. Reusable integration accelerators address the gap by packaging these cross-cutting capabilities into components that many services can adopt without re-implementing the same logic independently. Three structural patterns dominate current practice: the microservice chassis, which supplies a shared foundation at build time; the sidecar, which runs cross-cutting logic as a companion process alongside each service instance; and the API gateway or service mesh, which centralizes concerns such as routing and policy enforcement at the network layer. Each pattern trades a degree of service independence for consistency and reduced duplication, and teams typically combine more than one depending on which concerns are addressed. Governance practices, including scoping accelerators to strictly non-domain logic and reviewing shared dependencies periodically, determine whether the resulting coupling remains manageable or becomes a bottleneck. The practical significance for organizations adopting microservices is that reuse decisions are not one-size-fits-all: the right accelerator for a given concern depends on whether that concern is better handled at build time, at runtime alongside the service, or at the network edge, and getting this placement right materially affects how fast teams can ship new services without accumulating inconsistency across the estate.

Keywords: Cross-cutting concerns; Microservices architecture; Reusable components; Service chassis; Sidecar pattern

1. Introduction

Microservices architecture divides an application into small, independently deployable services, each focused on a specific business function and interacting through clearly defined interfaces.

This approach offers significant benefits, such as enabling teams to develop, test, and release services on their own schedules and avoiding the complex dependencies that often slow down changes in traditional monolithic systems. A thorough examination of the architecture highlights a collection of interconnected patterns that address how services are broken down, how they communicate, how data is managed, and how they are deployed. These patterns are based on the lessons learned by teams who have encountered and resolved common challenges in distributed systems [1]. Reusability is a key part of these patterns. Several of them exist because similar issues appear regularly across different services, and solving them repeatedly would undo much of the advantage that decomposition is meant to provide.

* Corresponding author: Ganesh Kumar Gangannagari.

1.1.1. Reusability becomes a real issue once an organization has more than a few services in production.

Some logic varies from one service to another—such as the rules for a payments service versus a search service. However, other logic is almost the same across all services, like how a service authenticates a request, logs an error, or retries a failed call to another service. Treating these two types of logic in the same way is a common mistake early on. Organizations that automatically extract every repeated piece of code into a shared package often find that the package ends up taking on responsibilities it wasn't designed for, and changes to it can unexpectedly affect many services that depend on it [2]. Experience shows the importance of distinguishing between technical, infrastructure-related concerns that are suitable for a shared foundation, and business or domain logic, which should stay local to the service it belongs to, even if that means some duplication is necessary.

1.2. This distinction is central to the discussion that follows.

In this context, a reusable integration accelerator refers to any packaged component—such as a library, a helper process, or a shared infrastructure element—that a team can use to manage a technical, cross-cutting concern without having to write the logic themselves. The main benefits are fewer lines of code, more consistency across services, and faster onboarding for new services that can start from a trusted foundation rather than an empty codebase. However, the risk is also clear: any shared component becomes a dependency, and dependencies are something microservices architecture aims to minimize. Because of this, organizations tend to be cautious when reusing components, focusing on narrow scope and regularly reviewing them to ensure they haven't started to include business logic that should stay within the owning service.

1.3. Another important point is that reuse decisions are rarely permanent.

A capability that begins as duplicated code in two services may later justify being moved into a shared component once a third or fourth service requires the same functionality. Conversely, a shared component that becomes a bottleneck for releases might be intentionally split to restore service independence. For teams starting this process, the takeaway is that using a catalog of established patterns, rather than making arbitrary decisions, tends to lead to more predictable results. This is because each pattern in the catalog comes with a documented set of trade-offs, allowing teams to evaluate whether an approach fits their specific constraints before proceeding.

Table 1 Shared Code Reuse vs. Coupling Risk by Architectural Concern [1, 2]

Category	Typical scope	Reuse candidate	Coupling risk
Domain logic	Business rules specific to one service	Rarely	High if shared
Cross-cutting technical concern	Logging, authentication, tracing	Frequently	Moderate, manageable with scoping
Build and deployment tooling	Templates, pipelines, base images	Frequently	Low
Network-level policy	Rate limiting, routing, TLS termination	Frequently	Low, externalized from the service
Data access pattern	Persistence strategy per service	Rarely	High if shared

2. Cross-Cutting Concerns and the Coupling Trade-off

A cross-cutting concern refers to a type of functionality that impacts multiple parts of a system but does not fit neatly into any one specific component.

In a microservices environment, clear examples of such concerns are operational. Each service must handle authentication of incoming requests, track a request as it moves across different services, and log what occurs when something goes wrong. These tasks are not exclusive to any particular service's business logic, but they are necessary for all services to function properly in a production setting. This is why these concerns are considered cross-cutting rather than just common—their logic is not duplicated by accident, but is structurally required in every service because they apply consistently across the system.

This situation leads to a known challenge among developers who have faced it directly.

One approach, based on daily experience building shared packages, suggests that it is acceptable, and in some cases preferable, to repeat code across services rather than use a shared library that introduces a binary dependency between teams. Once multiple services rely on the same library, any change in that library requires coordination across all teams that use it, and this coordination increases as more services adopt the shared component. Based on this experience, the recommendation is to use shared libraries mainly for concerns that are unlikely to change due to business needs, like retry logic or connection pooling, while avoiding sharing domain models.

A different perspective, from the standpoint of tooling, suggests that many organizations avoid implementing cross-cutting concerns within application code entirely.

Instead, they move functions like authentication, monitoring, and request tracking into infrastructure that exists outside the service itself, such as a proxy or a special-purpose chassis. This approach ensures that changes to shared logic do not require updating or redeploying every service that uses it.

Both viewpoints agree on one key point: the coupling created by reusing code is not absolute; it varies depending on where the shared logic is placed.

If a shared library is compiled directly into each service's binary, it creates the strongest coupling because any change to the library would necessitate a rebuild and redeployment across all services. However, if the shared component runs as a separate process alongside the service and communicates over a local network, the coupling is significantly reduced since the shared module can be updated and redeployed independently. A shared policy enforced at a gateway or proxy layer further reduces coupling, although it may reduce visibility: developers working within a single service can no longer see all the cross-cutting behaviors applied to their requests simply by reviewing the service's code.

This trade-off is why organizations rarely choose a single reuse strategy for all cross-cutting concerns.

Instead, teams usually categorize each concern based on how often it changes, how closely it needs to interact with business logic, and how much operational insight developers need. Then, they select the least tightly coupled approach that still meets those requirements. For example, a concern that rarely changes and does not require per-service customization, such as TLS termination, is well-suited for being centralized outside the service. On the other hand, a concern that needs to be aware of specific request context but doesn't require a service to be redeployed, like structured logging format, is better handled by a companion process. The outcome is a layered approach to reuse rather than relying on a single shared library to manage everything. This method ensures that any one dependency's impact is limited and more manageable.

Table 2 Cross-Cutting Concerns: Coupling Trade-offs Between Shared Libraries and Externalised Services [3, 4]

Cross-cutting concern	Coupling level if shared as a library	Coupling level if externalized
Authentication and authorization	High	Low, centralized at gateway
Structured logging	Moderate	Low, offloaded to companion process
Distributed tracing correlation	Moderate	Low, instrumented at proxy layer
Retry and backoff logic	Low, stable and rarely revised	Low either way
Rate limiting	High if per-service	Low, centralized at edge

3. Governance Practices for Reusable Components

Deciding to extract a cross-cutting concern into a reusable component is only the first step; the harder and more durable problem is governing that component once multiple teams depend on it. Practitioner guidance that has accumulated around this problem converges on a small set of recurring rules, even though the specific vocabulary varies from one source to another. The first and most consistently repeated rule is scope discipline: a shared library should hold code that is unlikely to change for business reasons, and it should never hold domain models or business rules, because doing so ties the release cadence of every consuming service to changes that are, by definition, specific to only one of them [5].

Utility functions, cross-service HTTP client behaviour, and standardized error translation are cited as reasonable candidates precisely because they tend to be stable and are not expected to diverge across services.

A second recurring governance practice is treating a reusable component as a product with its own consumers, rather than as an internal implementation detail that any team can modify freely. This framing matters because a shared component used by only its original author behaves very differently from one consumed by a dozen services maintained by separate teams; the latter requires a defined interface, a versioning policy, and some mechanism for consumers to opt into upgrades on their own schedule rather than having a breaking change forced onto them without warning [6]. Where an organization has moved from ungoverned “common” packages toward a scoped, deliberately maintained framework, the reported benefit is a reduction in the kind of unplanned coordination overhead that erodes the deployment independence microservices architecture is meant to provide in the first place.

A third practice is periodic review of what has accumulated inside a shared component over time. Shared libraries tend to grow by accretion: a developer facing a deadline adds “just one more utility” to an existing package rather than starting a new one, and over months this pattern can leave a component carrying responsibilities well outside its original scope. Guidance on this point recommends treating growth in a shared dependency as a signal worth investigating rather than a neutral outcome, and periodically auditing whether functionality inside it still belongs there or has quietly drifted into domain-specific territory that should be pulled back out into the individual services that actually own it [5]. This review discipline is what distinguishes an accelerator that remains genuinely reusable from one that has effectively become an undeclared extension of every service that depends on it.

A fourth practice, more architectural than procedural, is to prefer packaging cross-cutting logic as a standardized, versioned foundation rather than as an ad hoc grab-bag of utilities. A structured foundation of this kind supplies a documented and testable feature set, such as service registration or configuration handling, that is stabilized once and then reused with minimal per-service testing burden, since a change to the foundation does not need to be revalidated inside every consuming service's own test suite in the same way an unstructured shared utility would [6]. This structured approach tends to reduce the ambiguity that surrounds looser shared libraries, because its scope and contract are established deliberately rather than emerging from whatever happened to be convenient to extract at the time.

Taken together, these governance practices describe reuse as an ongoing discipline rather than a one-time architectural decision. Extracting a shared component is comparatively easy; keeping it scoped correctly, versioned predictably, and free of business logic as an organization's services and teams multiply is the part that determines whether the accelerator continues to deliver a net reduction in effort or gradually becomes the source of the very coordination overhead it was meant to eliminate.

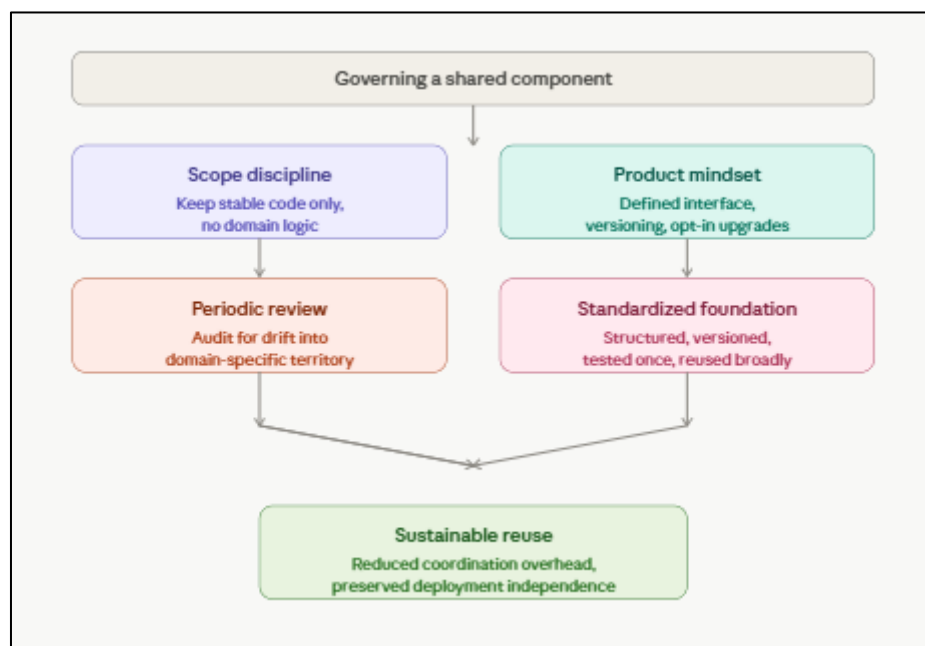


Figure 1 Governance practices for reusable components: four pillars of sustainable reuse [5, 6]

4. Architectural Manifestations of Reusable Accelerators

Reusable accelerators for cross-cutting concerns come in various distinct architectural forms, and the choice among them depends largely on where the shared logic is physically located in relation to the service it supports.

A basic definition of the architecture describes microservices as small, independently deployable units that communicate via well-defined interfaces, with each service focusing on a specific business capability so it can be developed and scaled on its own [7]. This independence is what makes the placement of shared, cross-cutting logic a purposeful design choice rather than an afterthought: placing logic inside the service's own process ties it to that service's release cycle, while placing it outside preserves independence, though at the cost of an extra network hop or an additional operational component to manage.

A commonly cited definition describes a cross-cutting concern as any aspect of a system that affects multiple components and is not limited to a single one.

It identifies aspect-oriented programming, middleware, and established design patterns as the main ways to separate such concerns from core business logic [8]. This approach is helpful because it applies across different implementation choices: whether an organization uses a shared library, a separate process, or a network-layer policy engine, the underlying goal remains the same—to isolate logic that does not belong to any one service's domain. This ensures that domain code remains clear and that each service's main responsibility is not hidden by infrastructure concerns spread throughout the system.

In practice, three architectural placements are most commonly used.

The first keeps shared logic compiled or packaged directly into the service, often through a foundational framework, sometimes called a chassis, that a team adopts when a service is created. This framework provides build tools, configuration handling, and basic cross-cutting behavior from the start. The second option places shared logic in a separate process that runs alongside the service instance, often deployed as a companion container that intercepts or enhances the service's network traffic without changing the service's own code. The third approach completely externalizes shared logic, enforcing it at a gateway or proxy that sits in front of multiple services at the same time. This gateway applies policies such as authentication or rate limiting before a request even reaches a specific service. Each placement handles a different set of concerns and has a different impact on operations, and most production environments combine more than one approach. They might use a foundational framework for build-time consistency, a companion process for request-level concerns that require closeness to the service, and a gateway for policies that are consistent across the entire system.

The common trend in these definitions is a reduction in the physical coupling between shared logic and the service it supports.

Table 3 Where shared logic lives: architectural placement options for cross-cutting concerns [7, 8]

Architectural placement	Where shared logic runs	Typical concerns absorbed
Foundational service framework	Compiled into the service at build time	Build tooling, configuration, service registration
Companion process	Separate process alongside the service instance	Logging, tracing, local retries, TLS handling
Gateway or edge policy layer	Outside any individual service, at the network edge	Routing, authentication, rate limiting
Middleware or interceptor	Woven into the request pipeline within the service	Request validation, response transformation
Aspect-oriented modules	Woven into code at defined points via a language mechanism	Method-level logging, transaction boundaries

It moves from logic embedded in the application code, to logic in a separate but nearby process, to logic enforced entirely outside of any single service's deployment unit. This progression mirrors the coupling trade-off discussed for individual

concerns, but at the architectural level overall. An organization's decision on where to place accelerators, rather than any singular choice, determines how much of its cross-cutting logic can evolve independently of the business services it supports.

5. Canonical Patterns: The Chassis and the Sidecar

Two named patterns capture most of what practitioners mean when they refer to a reusable integration accelerator: the microservice chassis and the sidecar. Both address the same underlying problem, the proliferation of cross-cutting concerns across many independently deployed services, but they solve it at different points in a service's lifecycle and with different operational trade-offs.

The chassis pattern calls for assembling a foundational framework that a team creates once and then reuses as the starting point for every new service. This foundation typically bundles reusable build logic, such as the plugins needed to build and test a service consistently, alongside mechanisms that handle cross-cutting concerns like service registration, client-side service discovery, circuit breaking, and distributed tracing instrumentation. Rather than assembling this collection of frameworks and libraries independently for every new service, a team builds and stabilizes the chassis once, and each subsequent service starts from that stabilized foundation. The stated advantage is that keeping dependencies, build logic, and cross-cutting behaviour current becomes far simpler: an update is released once in the chassis itself, and each service adopts the new version at its own pace by upgrading its dependency, rather than an engineer manually re-implementing the same fix across every service individually [9]. The chassis pattern relates closely to several other patterns in the same catalogue, since it is often the layer responsible for implementing self-registration, client-side discovery, and circuit breaking on behalf of every service that adopts it, and a service template is frequently offered alongside the chassis as a concrete example of a service already wired to use it.

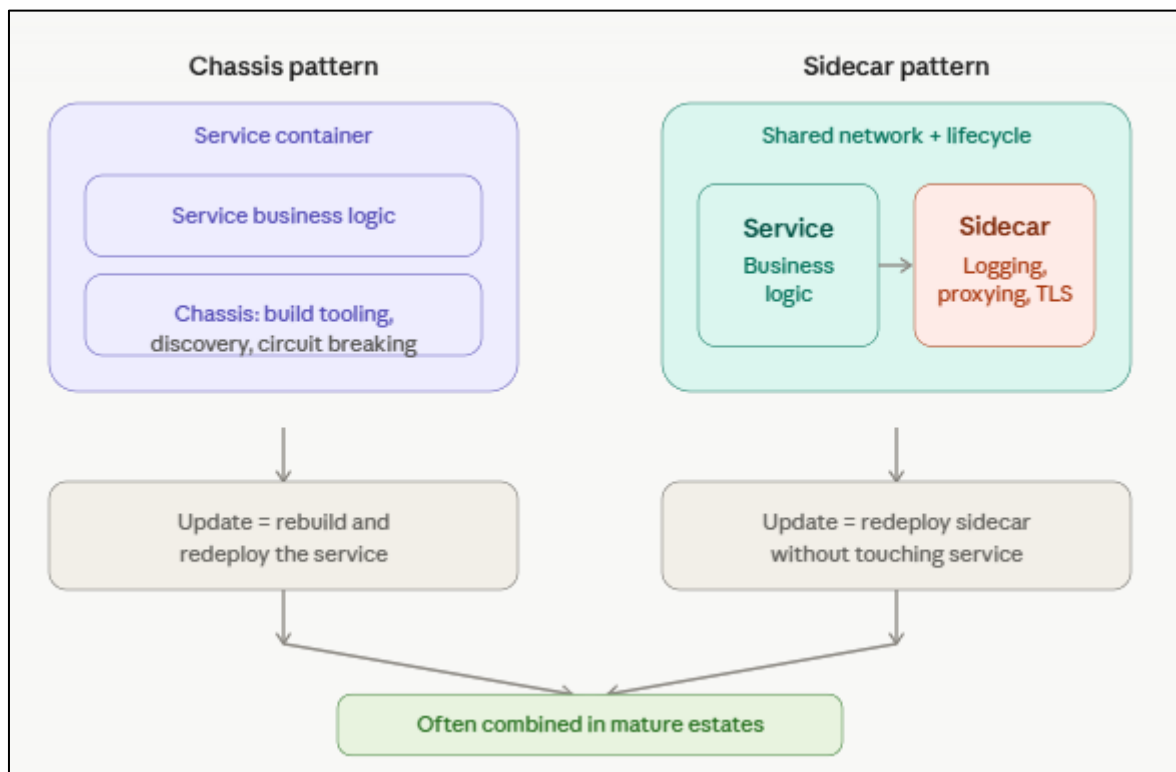


Figure 2 The chassis and the sidecar: two placements for cross-cutting logic [9, 10]

The sidecar pattern takes a different approach: rather than compiling shared logic into the service itself, it deploys a companion process, in modern deployments almost always a companion container, that runs alongside the main service instance and shares its network and lifecycle. The main service delegates specific cross-cutting responsibilities, logging, monitoring, proxying, and configuration updates among them, to this companion process rather than implementing that logic itself [10]. Because the sidecar runs as a genuinely separate process, it can be developed, tested, and even written in an entirely different technology stack from the main service, and it can be upgraded independently as long as the interface between the two remains stable. This is a meaningfully looser coupling than the chassis pattern provides, since

a chassis update still requires rebuilding and redeploying the service that consumes it, whereas a sidecar update can, in many deployment models, be rolled out without touching the main service's own container image at all.

The two patterns are not mutually exclusive, and mature microservices estates commonly use both together: a chassis establishes consistent build tooling and baseline service behaviour at creation time, while a sidecar handles concerns that benefit from being deployed and scaled independently of the service's own release cycle. Choosing between them, or combining them, comes down to the same question raised throughout this discussion: how tightly does a given cross-cutting concern need to be coordinated with the service's own code, and how much operational overhead is an organization willing to accept in exchange for looser coupling. A concern that is genuinely stable and rarely changes may be perfectly well served by a chassis; a concern that benefits from independent scaling, frequent updates, or polyglot flexibility across services written in different languages is usually better served by a sidecar.

6. Conclusion

Across the patterns and practices surveyed here, a consistent picture emerges: cross-cutting concerns are an unavoidable by-product of decomposing a system into many independently deployable services, and reusable integration accelerators exist to keep that duplication from eroding the benefits decomposition was meant to deliver. The central design question is never simply whether to reuse a piece of logic, but where that reused logic should physically live relative to the services it supports. Logic compiled directly into a service through a foundational chassis offers consistency and a fast-starting point for new services, at the cost of tying updates to that service's own release cycle. Logic delegated to a companion sidecar process loosens that coupling considerably, letting cross-cutting capabilities evolve on their own schedule. Logic enforced entirely outside any individual service, at a gateway or policy layer, loosens coupling further still, though it removes some of that behaviour from the immediate visibility of the developers working inside a given service.

References

- [1] Richardson, C. (2018). *Microservices patterns: With examples in Java*. Manning Publications.
- [2] Gulati, S. (2021, October 20). When to use shared libraries in Microservices architecture. <https://shekhargulati.com/2021/10/20/when-to-use-shared-libraries-in-microservices-architecture/>
- [3] Kanjilal, J. (2021, November 19). How a microservices chassis regulates cross-cutting concerns. TechTarget. <https://www.techtarget.com/searchapparchitecture/tip/How-a-microservices-chassis-regulates-cross-cutting-concerns>
- [4] Cusimano, S. (2022, November 9). Microservices and cross-cutting concerns. Baeldung on Computer Science. <https://www.baeldung.com/cs/microservices-cross-cutting-concerns>
- [5] Cline, B. (2023, January 4). When to use shared libraries in microservices. <https://www.brcline.com/blog/when-to-use-shared-libraries-in-microservices>
- [6] AppMaster. (2023, August 27). Microservices chassis. AppMaster Glossary. <https://appmaster.io/glossary/microservices-chassis>
- [7] Codefresh. (2023, September 25). What is a microservices architecture? <https://codefresh.io/learn/microservices/what-is-a-microservices-architecture/>
- [8] DevX. (2023, October 18). Cross-cutting concern. <https://www.devx.com/terms/cross-cutting-concern/>
- [9] Richardson, C. (n.d.). Pattern: Microservice chassis. Microservices.io. <https://microservices.io/patterns/microservice-chassis.html>
- [10] Richardson, C. (n.d.). Pattern: Sidecar. Microservices.io. <https://microservices.io/patterns/deployment/sidecar.html>