P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Legacy Decoupling Patterns for Health Insurance

Niklas Andersson
Asst Professor

Abstract

Monolith coupling in health insurance platforms impedes growth as business environments change. The Migration Strangler Pattern offers a decoupling strategy, but selection of additional patterns lacks empirical support. Qualitative architectural analysis identifies and describes a catalogue of 14 patterns that address remaining decoupling concerns across business areas such as observability and monitoring, deployment and operations, risk management, and team organization. A synthesis of nine architectural principles supports the patterns. Evidence from three health insurance case studies assesses pattern applicability, clarifies trade-offs, and expresses cost-benefit considerations.

Introduction

Monolith coupling in health insurance platforms limits scalable growth in increasingly volatile business environments. The migration-supporting Migration Strangler Pattern offers a single decoupling pathway, but additional patterns for related concerns across operational domains remain poorly substantiated. A qualitative architectural analysis identifies and describes a pattern catalogue covering 14 additional transformation avenues associated with observability, deployment and operations, CI/CD, security and compliance, and team organization. The catalogue is framed by a synthesis of nine architectural principles for supporting decoupling outside direct functionality. Patterns are grounded in practical experience from three health-insurance-sector case studies that attest applicability, clarify trade-offs, and express cost-benefit considerations.

keywords: Microservices Architecture in Health Insurance,Strangler Fig Pattern for Legacy Systems,Domain-Driven Design (DDD) in Insurance Platforms,API Gateway Pattern for Monolith Decomposition,Event-Driven Architecture in Healthcare Systems,Service-Oriented Architecture (SOA) Modernization,Legacy System Refactoring Strategies,Data Decoupling and Database Segregation,Health Insurance Core System Modernization,Backend-for-Frontend (BFF) Pattern in Insurance Apps,Containerization and Kubernetes for Legacy Migration,Anti-Corruption Layer Pattern in Healthcare IT,Incremental Migration of Monolithic Applications,Cloud-Native Transformation in Insurance Platforms,Interoperability Standards (FHIR, HL7) Integration.

1. Introduction

Many health insurance platforms are built as monoliths, often over many years and through the contribution of large teams. Health insurance is subject to complex compliance requirements, often in different geographies, necessitating constant and broad change. Delivery speed has consequently become a business differentiator, leading to the emergence of Event-Driven

Architectures and Domain-Driven Design as approaches to decouple delivery and practical considerations. Nonetheless, although coupled monoliths are the problem, within large organizations with health insurance platforms these techniques may often not be sufficient to enable a decoupling with confidence and achieve organizational autonomy of teams.

A catalogue of architectural patterns is articulated to support the decoupling of monolithic platforms. Three questions guide the work: Which architectural patterns help decouple a coupled monolith in a health insurance domain? How can those patterns be synthesized to establish a cohesive decoupling strategy? How have patterns been implemented to establish confidence in decoupling? The contribution draws from an exploratory qualitative-design, using a case-study approach. Patterns emerge from industry experience and are validated with process-case evidence. Decoupling goals are an instinctive gauntlet being investigated within the catalogue patterns.

1.1. Background and Significance

Health insurance platforms typically implement business logic in a highly coupled manner. The gradual emergence of Event-Driven Architectures, Event Sourcing, Domain-Driven Design, and Partial Extract-Transform-Load processing addresses coupling, enabling the dissolution of monoliths into distributed systems without the risk of disrupting business operations. In practice, these techniques operate as preservation-oriented antipatterns to a modularization strategy that an enterprise executes at its own pace, using capabilities deployed by CI/CD pipelines. A Smalltalk translator built for European income tax systems provides a real-world case. The analysis is supported by a catalogue of architecture-and-data patterns, tailor-made for controlling risk in the decoupling of coupled systems and the gradual migration from classic database schemes driven by the Shared Data Ownership antipattern requiring Distributed Transactions.

Business domain transformation in the health insurance sector may yield unforeseen consequences. New regulatory requirements or media scrutiny may give customers a stronger bargaining position. Infectious diseases can affect claims ratios. Changes in the law may compel insurers to accommodate their clients' cash flow preferences. To reassure customers and creditors, an enterprise may wish to demonstrate a strong solvency margin by declining underwriting for an entire territory or business segment. Databases used by health insurance platforms deploy operational data that is necessary for companies to respond to business pressures.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

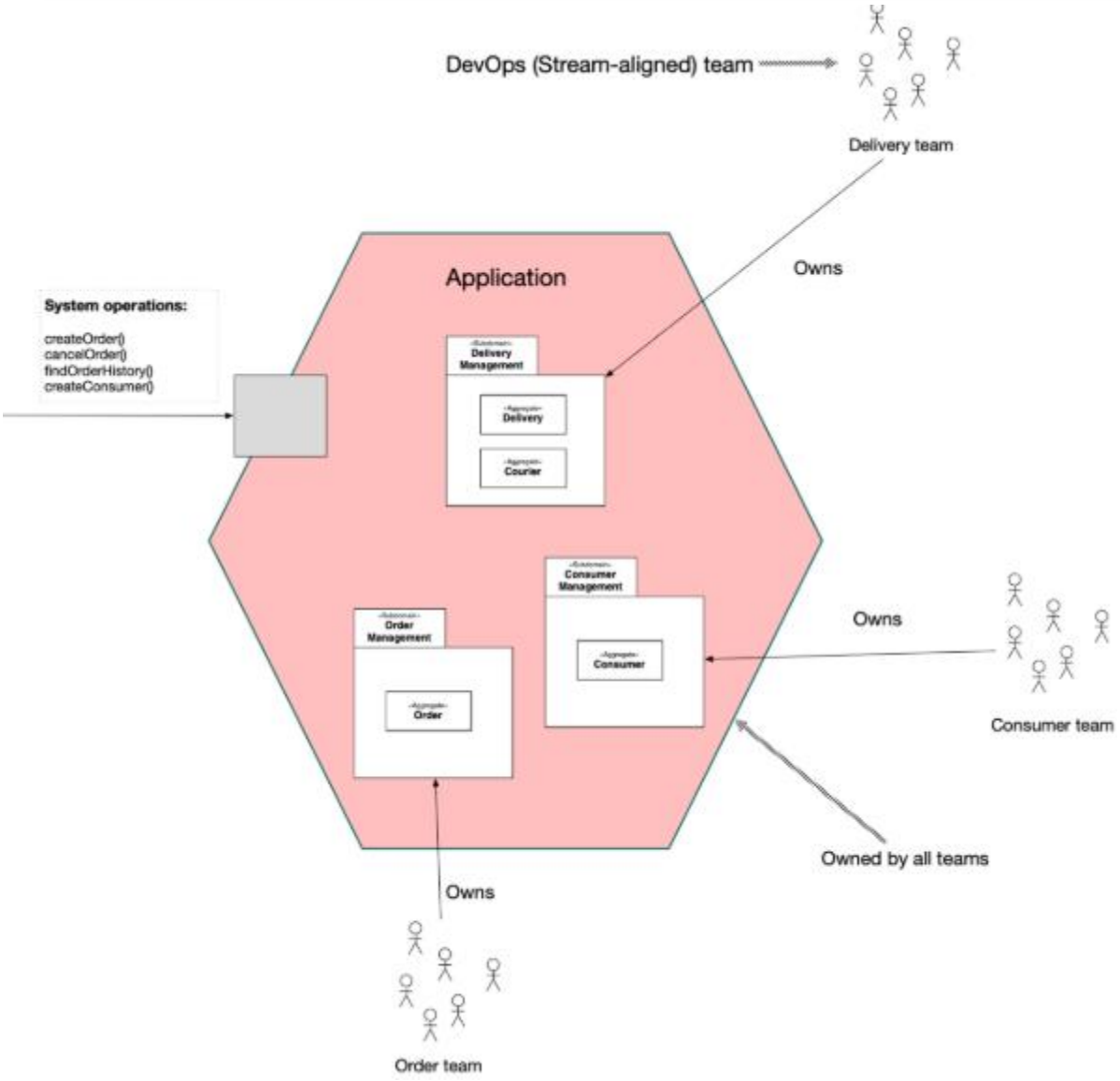


Fig 1: Pattern Monolithic Architecture

1.2. Research design

Design considers qualitative architectural discovery, evidenced by three health insurance case studies that inform ten overall

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

criteria distilled into a pattern catalogue. Selection prioritizes expected outcome value, while patterns require latest-art validation for successful implementation.

The architectural analysis prioritizes the qualitative discovery of patterns useful for legacy monolith decoupling in health insurance domains. To ensure adequate pattern coverage, three rich case studies illustrate how health insurance organizations navigate the decoupling challenge. The resulting evidence is distilled into ten patterns presented for integration into ongoing transformation efforts. Selection scopes the ancillary risk management considerations of such initiatives. Because decoupling while delivering both high-frequency releases and internally consistent domains is nigh-impossible, organizations must manage the cost-benefit trade-off across all changes and reconcile any resulting concerns during the decoupling process.

The analysis design naturally overlaps with an architecturally observable transformation and can thus rely on active case evidence, all industry relevant. Active case reference supports the search for patterns that address Modularity and Boundaries, Interface-Driven Design, and Data Ownership and Caching Strategies. Evidence strength is prioritized over pattern coverage, although selection does seek both breadth and depth across the broader Transforming Legacy Monoliths catalog. Pattern outcomes—pursued to restore long-overdue cloud adoption, and to improve operational and audit cost efficiency—converge on Decoupling the Monolith and Delivering Highway 2.0, along with clear experiential endorsements across the three active cases.

Equation 1: Coupling reduction under modularization

Let:

- C= total effective coupling
- n= number of modules
- d_{ij}= dependency strength from module $i$to module j

Then total coupling can be modeled as:

$$C = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} d_{ij}$$

Step-by-step derivation

1. A monolith has many inter-module dependencies.
2. Each dependency contributes some coupling amount d_{ij}.
3. Total coupling is the sum of all pairwise dependency strengths.
4. Therefore,

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

$$C = \sum_{i=1}^{n} \sum_{j \neq i} d_{ij}$$

If decoupling removes some dependencies, say Δd_{ij}, then new coupling is:

$$C' = \sum_{i=1}^{n} \sum_{j \neq i}^{n} (d_{ij} - \Delta d_{ij})$$

So reduction is:

$$\Delta C = C - C' = \sum_{i=1}^{n} \sum_{j \neq i}^{n} \Delta d_{ij}$$

2. Background and Motivations

Healthcare is an ever-evolving field that presents deep-rooted challenges in software development. Transforming organizational goals and frequent, unpredictable policy changes require technology that supports fast, uncomplicated product re-configuration. Nevertheless, many insurance organizations around the globe are hamstrung by heavily-coupled monolithic systems that are difficult to replace as a whole. The architectural patterns catalog presented in this paper provides a structured approach to safely reconfigure monolithic legacy systems using existing and newly developed services.

The catalogue is informed by a qualitative analysis of architectures used to circumvent monolith coupling in health insurance platforms around the world. Suggested patterns are focused on architectural aspects explicitly related to module and service coupling. Empirical data on attempts to decouple these legacy systems is synthesized into principles and classified into a concise set of twelve patterns that address key aspects of decoupling and help manage the challenges. The resulting catalogue not only helps organizations identify ways of breaking free from monolithic coupling, but also assists in the implementation of those changes.

Aspect	Option	Advantages	Disadvantages
Data Ownership	Shared Database	Simpler integration	High coupling

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Aspect	Option	Advantages	Disadvantages
	Private Database	Independence	Eventual consistency
Data Access	Direct Read	Fast access	Tight coupling
	API-based Access	Decoupled	Latency overhead
Consistency Model	Strong Consistency	Reliable data	Reduced scalability
	Eventual Consistency	Scalable	Temporary inconsistency
Communication	Sync (Request/Response)	Simpler logic	Blocking
	Async (Events)	Loose coupling	Debug complexity

Table 1: Data Architecture Decisions

2.1. Event-Driven Architectures and Event Sourcing

Event-Driven Architectures harmonize an application's processing model with its data and business models, creating a scalable and responsive structure. Instead of storing data and processing requests in a single store-and-forward implementation, relevant business state changes trigger data storage and processing across loosely coupled services. Event Sourcing is a pattern that naturally emerges in Event-Driven Architectures when the data model consists of business state change events. As an alternative to using the traditional model of a snapshot of the current state stored for fast reads, the business state is reconstructed from the business state change events.

Health insurance platforms exhibit varying degrees of coupling across business capabilities and data. Event-driven decoupling efforts naturally flow from recognizing that a business capability consumes or produces events that mark significant changes in an underlying process: for example, "the product has been updated" or "the price for a service has changed." The hollowing out of the core is valid when there are external consumers of these events at a consistent and acceptable level of fidelity. It becomes untenable when parts of the application need to incorporate a lot of intricate processing to create or consume these events or must maintain data consistency across multiple business capabilities that produce and consume events at either extreme of the real-time spectrum.

2.2. Domain-Driven Design and Bounded Contexts

Both Event-Driven Architecture and Domain-Driven Design decoupling strategies rely on a deep understanding of an application domain, its data, distribution of responsibility in its operations, and challenges with communication and synchronization among subsystems. Domain-Driven Design provides systematic techniques for discovering the logical units in a domain and a process for creating a ubiquitous language across a domain to improve collaboration between domain experts and software engineers. The primary concepts of these techniques are Bounded Contexts, which identify the natural boundaries for a subsystem, strategic design, and ubiquitous language.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Within the strategic design process, the application domain is distilled into a set of Bounded Contexts. Bounded Contexts correspond to the natural boundaries of subdomains of the insurance business. Each Bounded Context should ideally align with a Module for which Module Ownership has been determined. Segregation of the business problem space into these subdomains reduces the cognitive load for the teams involved and enables concurrent developments. The integration of changes across Bounded Contexts requires careful planning and should be kept to a minimum. The concepts of Domain-Driven Design can help to identify subsystems that can be built and released as small independent systems that are horizontally segregated from the monolith and that have smaller integration surface areas with the monolith.

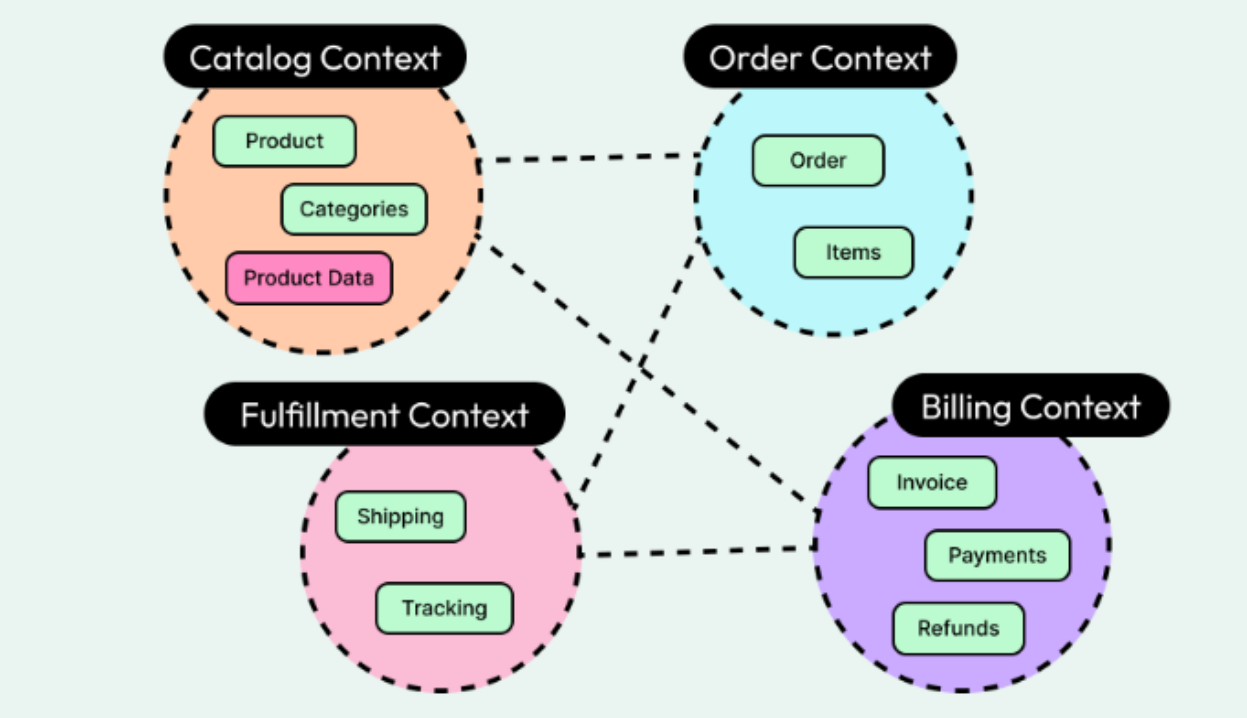


Fig 2: Domain-Driven Design (DDD)

2.3. Modular Monoliths and Feature Toggles

Modular Monoliths and Feature Toggles enable the gradual decoupling of monolithic health insurance platforms. Jointly, they facilitate the migration of domain capabilities to new infrastructures while downtime is avoided by toggling business processes on or off at runtime. Monolithic systems are grouped into modules based on Resilience Engineering principles. The modules may be mostly decoupled with downstream modules calling upstream modules synchronously, ownership of shared data may be transitioned, and data-consistency guarantees for virtually all data accesses defined. On a per-module basis, Feature Toggle patterns support the step-wise implementation of Strangler Fig sequences with ideally no code freeze, minimal downtime, limited additional risk compared to a big-bang migration, and measurable migration milestones.

A Modular Monolith is the logical decomposition of a single executable into distinct modules that implement the layers of an Event-Driven Architecture—presentation, orchestration, business process, domain logic, and data access functionality split into a number of sub-components or microservices—while remaining logically coupled for deployment, versioning, and data ownership purposes. The modules may be mostly decoupled, with Resilience Engineering principles determining the degree of coupling that is appropriate. Typically, it is foremost that a module does not directly call upstream modules, with downstream modules calling upstream modules synchronously only if it results in a clear efficiency gain.

Feature Toggles allow for the gradual implementation of the Strangler Fig pattern. They have a long history in CI/CD, allowing the merging of code that effects changes and is guarded by a toggle. However, they can also be exploited to contain the risk associated with the Strangler Fig pattern. By constructing the replacement implementation—or part of it—in new code that is under the control of a toggle, a business process can switch between the old and new implementation at runtime. Downtime for the switch can thus be avoided, testing can be performed by simply turning the toggle on, and further development can continue in the new infrastructure with very little risk. A properly managed toggle can even completely isolate the additional risk of the Sequencing pattern, making it introspectively quantifiable.

Equation 2: Risk reduction in Strangler Fig migration

Let:

- R_0= initial migration risk
- m= number of completed milestones
- r_k= risk reduction from milestone k

Then residual risk after $m$milestones is:

$$R_m = R_0 - \sum_{k=1}^{m} r_k$$

Step-by-step derivation

1. Start with initial total risk R_0.
2. Each milestone replaces a small function and lowers uncertainty.
3. If milestone $k$reduces risk by r_k, subtract that amount.
4. After $m$milestones, subtract all milestone reductions:

$$R_m = R_0 - r_1 - r_2 - \cdots - r_m$$

5. In sigma notation:

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

$$R_m = R_0 - \sum_{k=1}^{m} r_k$$

If each milestone has equal impact r, then:

$$R_m = R_0 - mr$$

3. Architectural Principles for Decoupling

Architectural principles support the previously proposed decoupling patterns. Modularity and boundaries establish separation criteria and coupling metrics. Interface-driven design champions API-first development, contract testing, and versioning strategies. Data ownership and caching strategies encompass ownership models, cache invalidation, and read/write patterns.

Modularity is an established principle of software design. Good modularization actively promotes change, isolates fragile or volatile elements, clearly separates responsibilities, and minimizes dependencies. Yet monoliths frequently remain modular only in theory. Concerns about customer-facing functionality often take precedence over good architectural practices. Even casual or aspiring developers may introduce poor design by plunging into straightforward coding instead of spending the time accurately defining proper software modules. Integration for integration's sake promotes high coupling between modules. Features with weak business cases may further motivate artificial coupling. Performance and complexity concerns exacerbate these issues. Ultimately, the principle is that modules should have low coupling and high cohesion. Actual decoupling, coupled with testing, scripting, and monitoring, can even allow milestones around broken features, product freezes, and ad-hoc coupling rechecks to be dispensed with during a Strangler Fig migration.

Interface-driven design develops systems using external first approaches and contract testing. API-first development encourages investments to create and maintain a specification and contract testing ensures that conjunctive API guarantees hold through change. Such specification languages should be user-friendly. Library support should simplify mock generation. Assuring that service consumers can adapt to and test against a new service version ahead of time should mitigate versioning risk. Versions should be time-limited on the service provider side and should not be eliminated until consumers have adapted. API Gateways also centralize security concerns. These advantages drive the often-encountered feature of API gateways. Backend-for-frontend architectures resemble API gateways but fine-tune routing for specific presentation-layer needs, harmoniously integrating telemetry, monitoring, and logging concerns.

3.1. Modularity and Boundaries

A fundamental principle when approaching decoupling is that parts of the coupled system should be identifiable as modules with clearly defined boundaries. The more distinctly separated, the easier it is to deploy, replace, and change modules independently of one another. While the solution space spans multiple layers of the architecture, those at the design layer provide strong guidance. Within the context of Domain-Driven Design (DDD), the guiding principle has led to the notion of Bounded Context. A Bounded Context contains a well-defined part of the business solution and is augmented with a dedicated part of the data

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

model. The language used within the context is carefully illustrated to promote Ubiquitous Language, which reduces ambiguity and inconsistency. It is therefore suitable for avoiding translation problems across business areas, facilitating location within the model, and supporting effective communication among stakeholders.

In an application where total dependability is difficult to achieve, the logical combination of different dependability strategies at different levels creates a better overall effect. Consequently, the interdependence among different subfunctions can be temporally and spatially relaxed. The Routing Approach decouples Invoke Time and Response Time between the Caller and Service Providers. The calling program only needs to send a request, and the main program can continue performing other tasks without having to wait for a request. The Backward Recovery Approach makes use of a shorter recovery time in order to relax the performance constraint between the Single Service Provider and the Dependent Service Provider. The Short-Circuiting Approach relaxes the data-consistency constraint that the operation sequences of two basic subfunctions have to be identical. The variation of the normal invocation time allowed by this approach may increase the total invocation time.

3.2. Interface-Driven Design

Decoupling a legacy monolithic application from one or more external systems often takes the form of API-driven integration waves. The prospect of coupling the legacy monolith to decoupled systems ought to be treated with caution, however. It is often - as with many locational stability discussions - global interfaces and APIs that are governed poorly and become a bottleneck for releases, creating incentives to bypass change through rapid spaghetti code integration around the edges. An API Gateway pattern or Backend for Frontends to feed frontend presentation layers, when driven API-first from Engineering, can help avert these problems. All systems integrations should be treated as a necessary Evil and done API-first with contract testing for those interfaces in place. API version management and deprecation must factor into any decisions so as not to become a hidden sinkhole. Furthermore, migrating towards a well-defined and tracked external contract helps prevent coupling simply to make it easier to consume this service from different systems, which is usually an indication that the API is too expansive in its surface area. APIs must be a limited shared interface that is not frivolously consumed.

Data ownership is another essential concept in decoupling a legacy monolithic application. APIs are just a view into the data stored temporally at excessive volume in shared databases. For read-heavy integrations, caching optimally takes effect; for write-heavy processes, cache invalidation and state synchronization back towards the database remain a formidable and elegantly unsolved engineering problem. Caches and materialized views are strategic components of any design; keeping state across systems synchronized with other technologies and simply routing back to the copy on the actual owner is just as useful a pattern, but one with a fundamental bias towards complexity when lots of state must be preserved and read ratio takes precedence.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

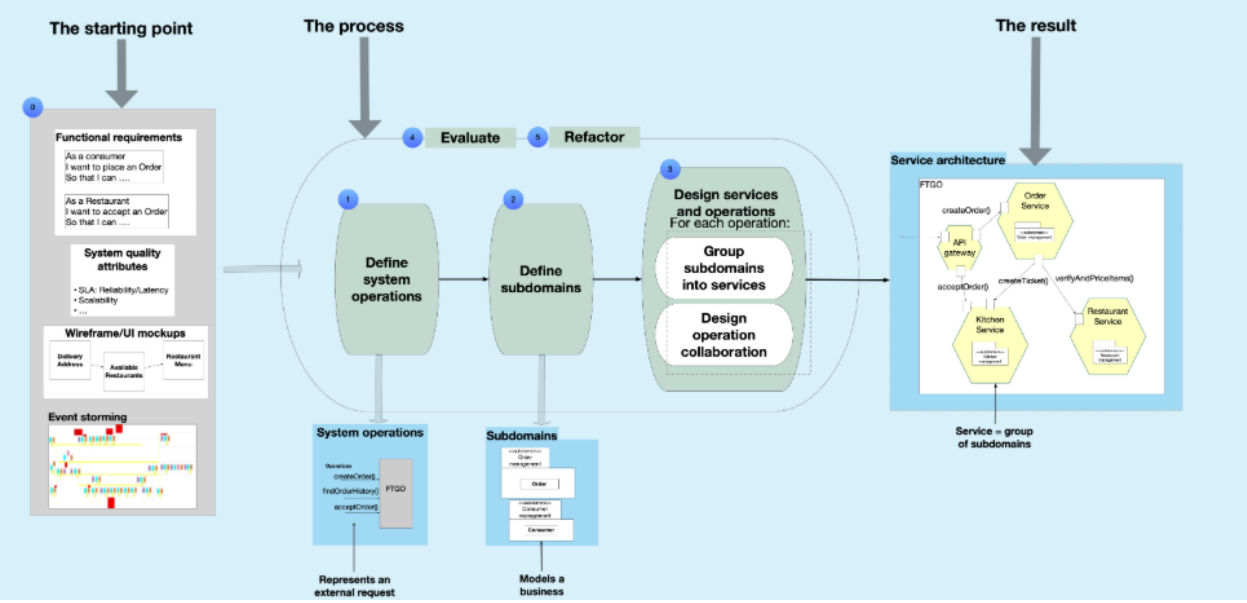


Fig 3: Interface-Driven Design of Architectural Patterns

3.3. Data Ownership and Caching Strategies

Data ownership decisions directly influence the degree of coupling between modules. A shared database architecture entails being aware of which software system owns each table, column, row, or field in the shared database. As data is created, read, updated, or deleted, application owners must understand their responsibilities for maintaining data consistency and correctness. Shared data often introduces significant time lags for committed data to become findable by systems that depend on it. If a separate module requires a particular piece of information frequently enough, implementing a local cache for that information in the dependent application may often be justifiable. Cache invalidation remains a crucial challenge for implementing any caching mechanism, especially when using a shared database.

Changing the data ownership model changes the guarantees of data consistency. Moving from a shared to a private data ownership strategy reduces coupling and coordination between modules at the cost of providing only eventual consistency guarantees. Event-driven architectures synchronized data at the time of creation or modification but still required policies and procedures for data lineage and data governance verification. When changing data ownership to a private model using an event-driven architecture, possible read patterns for the data can drive whether to publish the domain event using an asynchronous publish–subscribe mechanism or via a point-to-point solution. When the write side of an application is the primary producer of data events, an asynchronous solution is ideal for providing single data ownership with less coupling. If the read path is the primary producer, a command-query separation approach could be better suited to avoid the time lag.

Equation 3: Feature-toggle risk isolation equation

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Let:

- P_f = probability new feature fails
- I_f = impact if it fails
- T = toggle control factor, where $0 \leq T \leq 1$
- $T = 1$means perfect isolation, $T = 0$means no isolation

Expected feature risk without toggle:

$$ER_0 = P_f I_f$$

With toggle isolation:

$$ER_t = (1 - T)P_f I_f$$

Step-by-step derivation

1. Basic expected risk is probability times impact:

$$ER_0 = P_f I_f$$

2. A toggle reduces exposure because the feature can be turned off quickly.
3. Let Trepresent the fraction of risk isolated.
4. Remaining fraction is $1 - T$.
5. Therefore new expected risk is:

$$ER_t = (1 - T)ER_0$$

6. Substitute $ER_0 = P_f I_f$:

$$ER_t = (1 - T)P_f I_f$$

4. Pattern Catalogue

Decoupling a monolithic application into a distributed architecture can be a daunting task, often seen as a “big bang” migration. However, when handled properly, such decoupling becomes painless and fast, supporting agility within the organization. This

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

section presents several architectural patterns that consider various technical facets: data architecture, deployment and operations, compliance burden, cross-cutting concerns, and associated risk factors. These patterns have been derived from multiple independent case studies and author experiences, individually or in combination, guiding the gradual and safe transformation of major insurance platforms.

Most of these patterns support a gradual migration, enabling the co-existence of legacy and newly created systems. Such an approach is best expressed by the Strangler Fig pattern, while the Microservice Refactoring pattern focuses specifically on decomposing a microservice monolith. Other patterns provide supporting and controlling structures: the API Gateway controls access to a system from external clients, and the Backend for Frontends pattern creates an application layer that provides a tailored API suitable for specific consumer needs.

Strategy	Description	Risk Level	Speed	Use Case
Big Bang Migration	Replace entire system at once	Very High	Fast	Rare, high-risk cases
Incremental Migration	Gradual replacement	Low	Moderate	Preferred approach
Strangler Pattern	Replace features step-by-step	Low	Controlled	Legacy modernization
Parallel Run	Old + new run together	Medium	Slow	Validation scenarios

Table 2: Migration Strategy Comparison

4.1. Strangler Fig Pattern

The Strangler Fig pattern describes an approach for gradually replacing a legacy application with a new application. A small portion of the original functionality is replicated in the new application. Because both applications are supported during the migration phase, risk is mitigated. The name derives from the tropical Strangler Fig tree that grows slowly around its host until it can independently live in its environment.

Incremental migration is possible only if the two applications can coexist without irreconcilable differences. Functions in the legacy application are retired not when the new function is deemed finished but when the pressure of the remaining function has slowed. Migration speed is dependent on the organization's resource allocation to the change. For organizations that can afford to continue adding features to the original application or devote a large number of resources to the new one, the migration process can be measured more by risk than by time.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Once the temptation to rush an untested function to production is removed, migration speed can be managed by establishing measurable milestones. Replacing too large a portion of functionality creates a disaster waiting for a place to happen. Milestones allow a section of the duplicate functionality to be completed and entered into production before attention shifts to the next section. During normal operations, throughput is monitored to determine whether any part of the new function is causing degradation. Each completed milestone reduces the workload pressure on the legacy application. Clearly defined and easily measurable milestones address – if not resolve – the migration risk's biggest contributor: uncertainty. They allow the organization to measure its reactions rather than be governed by them.

4.2. Microservice Refactoring

Uncoupling a business capability previously built as a monolithic service can yield orbiting microservice teams that take full ownership of their service. To reduce risk, a business capability is upstreamed to a new team in a controlled manner so that it is still fully owned and operated by the legacy monolith but started to be consumed by the microservice. Once all functionality is covered and tested, the final upstreaming step takes place: the last pieces of the business capability are processed by the microservice.

There are three possible drivers for microservices refactoring:

1. Transform a distinct business capability, e.g., consumer ad campaigns, that has grown large enough to require a dedicated team. Data transfer models can be distilled.
2. Enable several teams that are responsible for an end-to-end business capability but are currently organized across several monolithic services to develop using a microservices strategy.

3. Migrate the organization towards a microservices governance strategy because of the consumption risks of a large monolithic service.

The boundaries of the microservice should then consider Has-and-Uses diagrams that establish the service relations for the business capability upstream and downstream within the organization.

Governance questions include whether three or four layers are warranted. A four-layer approach naturally allows the business people, who are too remote from development but still require the monolith as a partner, to define the proper business capability monitoring for upstream and downstream actions.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

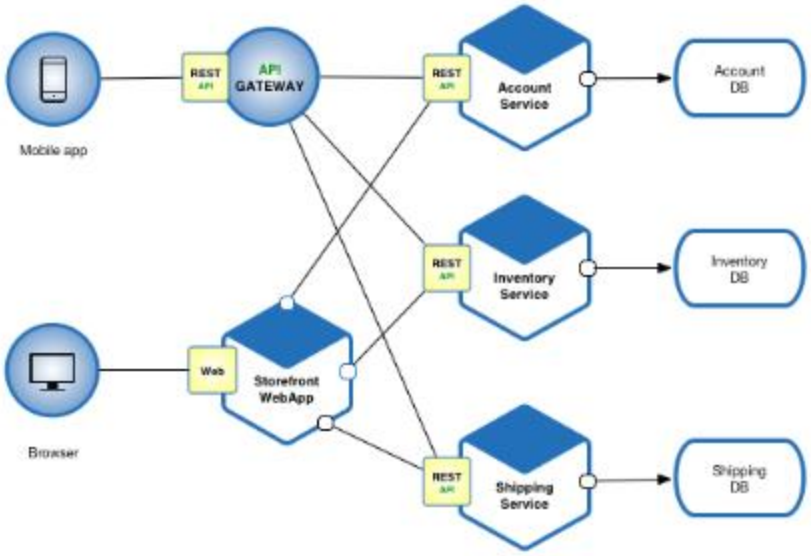


Fig 4: Microservice Architecture pattern

4.3. API Gateway and Backend for Frontends

An API Gateway pattern introduces a dedicated routing layer serving as a single entry point to various backend services. Such a layer centralizes security, simplifying the overall system's security posture by consolidating and hardening the API surface in one place. Moreover, an API Gateway typically incorporates cross-cutting concerns such as authentication, logging, and metrics in a single place.

Although routing is generally conceptually agnostic to the type of client consuming an API, there are often differences based on the client type. Routing functionality may be isolated behind a Backend for Frontends (BFF) service to cater to such discrepancies. The Gateway service then only handles authentication and authorization checks at the API surface; it does not need client type awareness.

When routing requests from clients, both API Gateways and BFFs must ensure the presence of role-based access control (RBAC) within each backend service, restricting access to the underlying services based on the client's role or group. The role of the client is determined through its identity provider and should propagate from the Gateway or BFF to the backend services. Each backend service should then deny access for any roles or groups that do not match its access policy.

Similarly, telemetry, tracing, and dashboards should be incorporated into the Gateway or BFF and be abstracted away from the backend services. Such concerns should be monitored centrally, allowing the product teams to focus on functional development. The incident response playbooks need to include triggers when incidents occur, along with information on who to notify based on the incident type.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Equation 4: Data ownership trade-off equation

Let:

- K= coupling
- L= synchronization latency
- E= eventual consistency penalty
- S= shared ownership indicator
- P= private ownership indicator

Model:

$$K_{\text{shared}} > K_{\text{private}}$$

and

$$E_{\text{shared}} < E_{\text{private}}$$

A simple weighted architecture cost function is:

$$A = \alpha K + \beta L + \gamma E$$

Step-by-step derivation

1. The architecture is affected by more than one factor.
2. The paper names at least coupling, latency, and consistency trade-offs.
3. Represent overall burden as a weighted sum:

$$A = \alpha K + \beta L + \gamma E$$

4. Under shared ownership:
    - coupling is higher,
    - consistency penalty is lower.
5. Under private ownership:
    - coupling is lower,

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

- consistency penalty is higher.

So compare:

$$A_{\text{shared}} = \alpha K_s + \beta L_s + \gamma E_s$$
$$A_{\text{private}} = \alpha K_p + \beta L_p + \gamma E_p$$

where

$$K_s > K_p, E_s < E_p$$

5. Data Architecture Considerations

For incremental decoupling, patterns and principles guide design decisions for authentication and authorization services. Services that are primarily accessed from read operations are sometimes allowed a direct connection to the legacy database. In these cases, the coupled legacies introduce the associated Państève risks of a shared data source. Where services with write access must read domains that are under construction or have already been rebuilt, the API-first principles with contract testing and versioning offered by Interface-Driven Design support the usage of service mocks. When a data event, such as in Event-Driven Architectures, must be the sole read influence for the writing service, the data public-access restriction is lifted for that operation only.

The Strangler Fig Pattern enables migration of the legacy core business functionalities into microservices through a wrapping and sequencing method. With sufficient telemetry, the usage of the components is evaluated to identify what can be replaced next. The metadata generated through observability supports the identification of logical migration units, their sequencing through business cycles, and the control of deployment risk. Each completed substitution delivers a functional piece of the new environment that, even if redundant, reduces the risk of change. Milestones are established to constantly measure and evaluate the progress. A refactoring plan defines the different use cases behind the security posture of the Application Programming Interfaces surrounding the execution of the business service. Mutually exclusive use cases are presented to separate presentation layers, taking advantage of Backend for Frontends.

5.1. Shared vs. Private Data Ownership

Collaborations among decoupled services frequently depend upon exchanging data. The owner of the data has to determine whether the data can be made accessible for reading or whether that data has to be directly written to. Reading from another service's data store may make the consuming service dependent on the provider, which brings particular risk to availability, data leakage, delays, and consistency. The risks associated with using shared data for reading can vary considerably depending on the overall use case, and hence a risk-reward analysis is necessary before proceeding.

Data that is written by one service and read by others should be separated from those consuming services during writing. If data inconsistency is tolerable during the read, eventual consistency may simplify the architecture; asynchronous message passing is fit for purpose. The latency introduced by the propagation of the written data should be consistent with the consumers' Service

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Level Agreements (SLAs) and the precautions around data lifetime taken by the writing service. If the request flow pattern does not align with these characteristics, direct access to the data source should be considered, and the services involved should share a Team Data Ownership and take responsibility jointly.

Where one microservice continues to own the authoritative source of record for a particular piece of data, the rules in each transaction guarantee that that data can be modified without subsequent change. With a combination of cache invalidation strategies, this remains a necessary and sufficient way to handle change while still allowing for scaling and indépendance of read traffic. If shared data were to be route-modified externally, an overhead of "gracefully" managing the write behind cache would now need to be carried on. Consequently, the enforceability conditions on allowed changes become extremely constrained and are usually enforced by the product domain. The service owners enabling that route-modification capability for the sake of availability or scaling to read at scale assume the responsibility if data is lost or inconsistent.

Equation 5: Saga success/rollback equation

Let a saga have $n$steps, and:

- p_i= probability step $i$succeeds
- c_i= probability compensation works for step i, if needed

Saga success probability

If all steps must succeed:

$$P(\text{saga success}) = \prod_{i=1}^{n} p_i$$

Step-by-step derivation

1. Saga completes only if step 1 succeeds, and step 2 succeeds, ..., and step $n$succeeds.
2. Assuming independence:

$$P(\text{all succeed}) = p_1 p_2 \cdots p_n$$

3. In product notation:

$$P(\text{saga success}) = \prod_{i=1}^{n} p_i$$

Compensation reliability

If failure occurs and compensation is needed on all completed prior steps, approximate rollback success as:

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

$$P(\text{rollback success}) = \prod_{i=1}^{k} c_i$$

5.2. Saga Patterns and Distributed Transactions

Saga patterns and distributed transactions support business process decoupling through choreography or orchestration, but trade-offs exist. Coordination requires either the involved participants or a distinct context, rollback behavior needs specification, and error handling should account for non-reversible interactions. Such considerations typically evade attention in marketable solutions; therefore, a customized effort involving joint design discussions is inevitable. Yet, where unactionable, independent analysis and modeling of the non-compensatable steps help convey the associated implications.

A saga represents a distributed transaction across multiple microservices capable of handling partial failures. Using choreography, each service declares the sagas it provides and subscribes to others through standard publish-subscribe mechanisms. Alternatively, orchestration invokes participating services through a central orchestrator. In either mode, each service ensures its execution is idempotent, part of a compensating transaction, or both.

Using choreography, compensation follows completion; if orchestration is the chosen trade, the compensating transaction executes when rollback is triggered. Regardless of the approach, sagas pose a coordination burden that cannot always be placed upon the participants. Even where feasible, complete compensation may not be possible: some microservice activities cannot be undone. Such concerns warrant proper treatment to avoid misuse. Missing attention on choreographed sagas raises additional issues. Deletions are generally permanent; therefore, resilience at this level is relative. Likewise, provisioning a resource in one step and altering it in another summons attention; initial provisioning remains unavailable, and a fault at this point halts even subsequent modifications until a downtime window allows resurrection.

Expanding the lessons of the present and past, the discussion herein identifies potential pitfalls when trying to deploy and operate legacy transformation with CI/CD. Such systems are heavily dependent on single/team/organization knowledge of the legacy artifact. Each of these artifacts must be accounted for in the operation pipeline, with a special CI/CD path for them. An artifact with a deprecated feature must also contain a rollback path. The same control must go for an experimental feature introduced by a Feature Flag.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

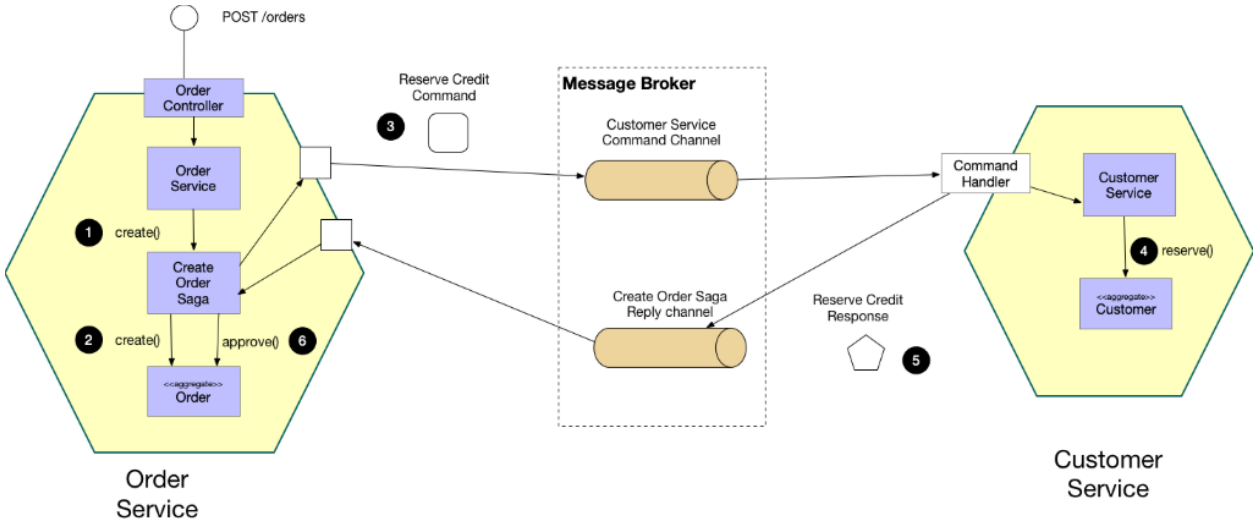


Fig 5: Saga Patterns

5.3. Data Migration Strategies

A planned data migration is critical when legacy services are replaced. A switchover cutover plan needs to be devised that will minimize service disruption, either by achieving all four aspects of a Big Bang cutover or by falling back on the legacy system. Metadata can assist in reconciling data and detecting data corruption, such as checksums for all account properties and summary statistics. Testing should include defining data reconciliation tests that cover both normal and edge conditions as well as how the target system can be resilient to data corruption during the migration. Automated tests of the reconciled state are ideal.

To ensure continuous evolution towards an optimal architecture, each migrated functional area should support data validation, validation tests should be automated as far as possible, and testing should be scoped to be realistic and cost-effective.

6. Deployment, Operations, and Governance

CI/CD pipelines underpin the delivery, deployment, and operation of modern systems at an unprecedented pace and scale. Many of these properties have led parts of the industry to express that "everybody wants CI/CD, but nobody actually wants to do it." Despite this, the banking and finance industry has many learnings in this discipline as a result of compliance and audit requirements.

All pattern trade-off conclusions are trade-offs. Here the points are summarised and compared with the risks and gains presented earlier. For each of them, the question, what happens when it is not taken, is discussed.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Security, compliance, and auditability are among the key topics that regulation tends to touch. In traditional banks, it touches almost every data aspect to some degree. It requires control on who accesses which data and for what purpose and regulation-driven telemetry of data but also of processes.

6.1. CI/CD in Legacy Transformation

Migrating a decade-old platform is a major IT project, and it requires a CI/CD strategy in order to enable rapid development while ensuring a high standard of quality. A separate team could take ownership of the Continuous Integration and Continuous Deployment (CI/CD) pipeline, but that would not be a good idea since such a complicated platform requires the involvement of almost all IT departments. It is therefore necessary for every department to have the opportunity to be part of the same team to ensure a smooth integration of such a CI/CD in all domains. Doing so guarantees that the artifact management repository is configured properly, that the created libraries, web services, and UIs are registered for auto-testing, that the code automatically undergoes all checks (no code smell, vulnerabilities, signup, and functional testing), that artifacts are automatically deployed in a testing environment and accessible for testing, and that a staging environment is created during the weekends to prepare for production rollout.

Feature flags (also referred to as toggles) are a well-known technique that allow teams to deploy code to production and then turn it on after testing. Care should, however, be applied when enabling feature flags in an existing codebase as they add extra complexity and increase the possibility of bugs being introduced. It is worth applying feature flags only when the risk of introducing a bug in production is considered too high to be taken, when monitoring is set up to detect the issue immediately, or when nothing is blocking the new code merge to production. A strategy and criteria should therefore be established to add, control, and end feature flags, not only for the current legacy transformation but also for the future evolution of the platform. Furthermore, complete rollback and recovery plans should be prepared for every release because enabling a feature flag that introduces a major defect must not take more than a couple of hours to correct.

6.2. Observability, Monitoring, and Incident Response

Every deployed service must be monitored and logs collected to support observability. The production environment constitutes the most complex deployment configuration and is usually a shared environment, hosting multiple services and functionally converging applications. Thus, distributed traces must be introduced to monitor the execution of user journeys. At the same time, each application must expose health and performance dashboards to support fire drills. Playbooks must be created to guide incident response teams resolving potential incidents.

Telemetry must be configured to populate an observability platform, enabling monitoring and alerting, while applications that respond to incidents must implement distributed tracing. The observability platform should populate a dashboard that makes cross-cutting information easily accessible and allow engineers to implement ad-hoc dashboards that group telemetry information according to specific needs. In addition to notifying of downtime, alerts should indicate situations of high errors and degraded performance. End-to-end test suites can run in a staged environment to validate the proper functioning of critical user journeys. Playbooks should suggest potential causes, workarounds, and procedures to investigate the issue for the most relevant incidents.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

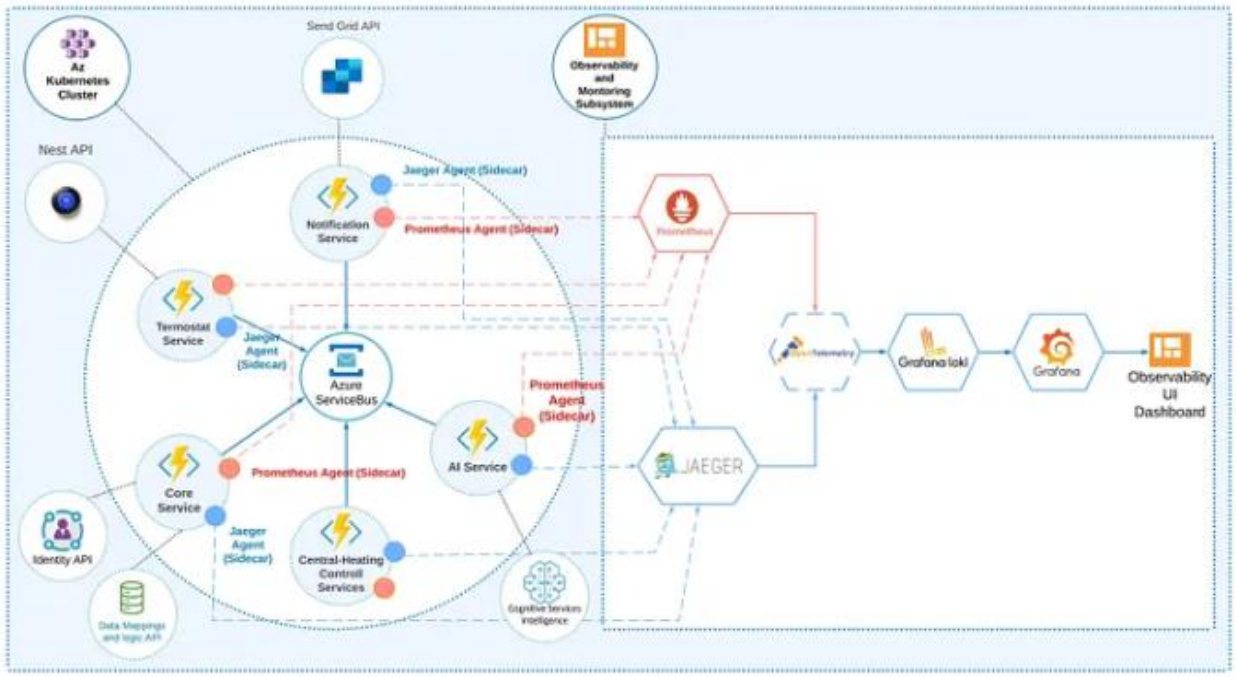


Fig 6: A Primer on Distributed Systems Observability

6.3. Security, Compliance, and Auditability

Compliance with regulatory requirements often necessitates access restrictions and sensitive data handling outside a system's standard security perimeter. A comprehensive security layer can support both external exposure and internal containment. Data privacy controls should be applied to detect leaks and unauthorized use of sensitive information. Data retention and archiving policies need to align with supporting regulations and be separately verified. Audit controls must meet regulatory demands while monitoring operations for unintended or harmful usage.

The overall security, compliance, and auditability posture of health insurance data systems is determined across the entire enterprise ecosystem and risk profile, as these factors affect tooling and viability. Teams should archive data outside business systems to ease compliance burdens from primary storage. Audit requirements may propagate back to the overall data ecosystem if certain events signal heightened concerns. Supporting privacy aspects can sharpen the incident-response approach and dynamically affect cloud language choices.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Dimension	Description	Impact
Performance	System speed	Affected by distribution
Consistency	Data correctness	Reduced in async systems
Availability	System uptime	May drop during migration
Cost	Financial investment	High for transformation
Complexity	System difficulty	Increases with microservices
Time-to-Value	Speed of benefits	Slower in phased migration

Table 3: Risk & Trade-off Dimensions

7. Risk Management and Trade-offs

Careful trade-offs are needed across risk factors for Performance, Consistency, and Availability as acceleration paths are defined for decoupling into microservices, feature flags are deployed for manageable pieces of change, and new features are built for experimentation without needing to complete the move. Patterns such as the Strangler Fig may impact how strictly business and technical teams have adhered to safety nets of CAP Theorem principles; for example, such patterns might make CAP guarantees less extreme during integration timeframes and become less important after decoupling is complete. Prioritizing changes affecting not only speed but also speed to market, observation and diagnostics, and resilience should be sought.

The Cost, Complexity, and Time-to-Value for a path to a high-declarative-roadmap adaptability, even post-completion of major decoupling steps, are important considerations for sustainability. Acknowledging and monitoring Preparation Effort, Downstream Dependence, Downstream Innovation Enablement, Development Skillset Change, Testset Size, and Resolution Elaboration Rates relative to a transformation goal assist in mitigating the risk of technical overload unnavigated by organizational leadership. Further change consolidation through reversible data consistency and less systemically affecting concepts enabled by Principle of

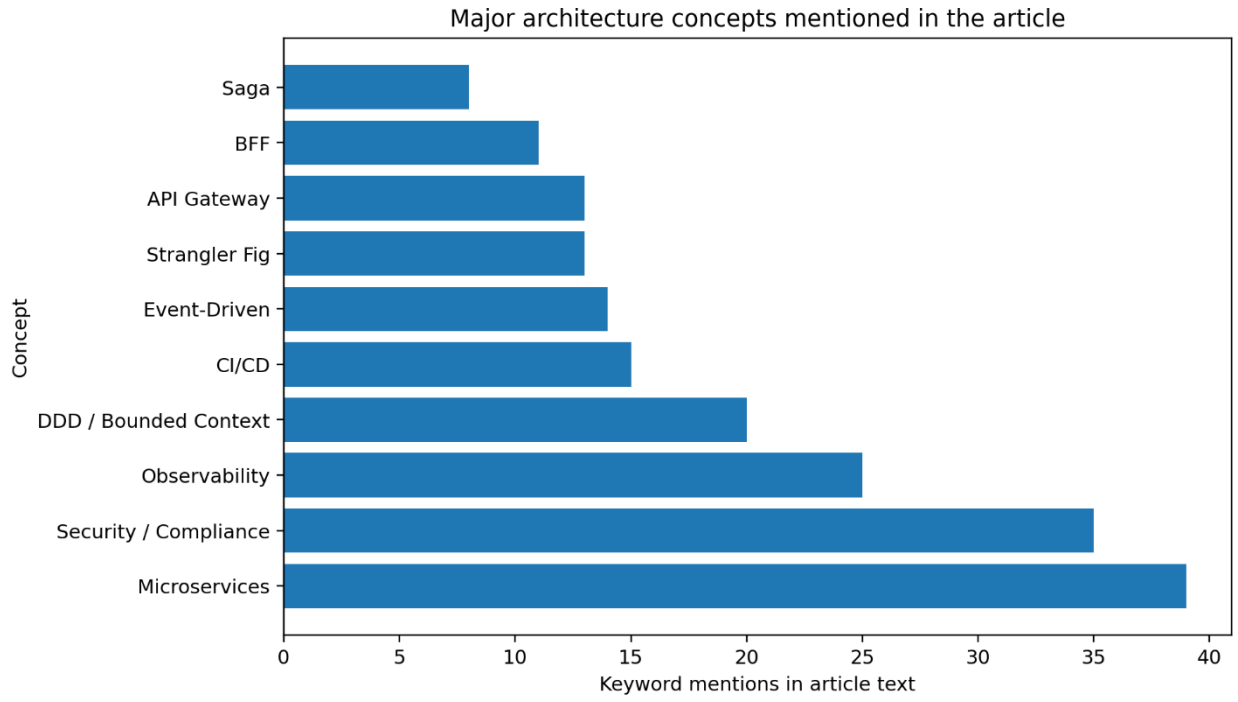
P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Least Interest should also be continually monitored. Regulatory audits or legal investigations introduce distinct and demanding requirements (representing Key Focus Acquisition).

7.1. Performance, Consistency, and Availability

The CAP theorem states that distributed systems cannot guarantee all three of the following properties simultaneously: Consistency, Availability, and Partition Tolerance. If a network partition occurs, at least one of the other two properties must be sacrificed. Within that trade-off space, different patterns support different balances, trading away Availability or Consistency depending on use case needs. For instance, a Strangler Fig migration can temporarily increase coupling and diminish availability as subsystems receive divergent code paths. In the context of Event-Driven Architectures, eventual consistency of state across the data topology may defeat user expectations during critical transactions, while the availability of replicated data sources naturally reduce performance and operational complexity.

Placing additional focus on consistency also introduces challenges. Choreographed saga patterns rely on limited misalignment within the business model and generally fail with major disruption events, as logic A copies variable X and logic B alters it in a window of time. Supporting these cases often requires longer-running orchestrated patterns resembling distributed transactions, which do recreate some form of strong consistency at the price of performance and complexity. While a single internal data store for a pure business capability can normally favour simplicity and speed of development, distribution often becomes necessary for paged or many-to-many integrations upon growing user bases.



7.2. Cost, Complexity, and Time-to-Value

Patterns simplify incremental refactoring of large, coupled systems, yet any decoupling initiative involves significant investment. Resources can dwindle at any stage, particularly for health insurance domains that must balance the needs of changing the organization with continuity of regulatory, financial, operational, and reputational audit-readiness. Evaluating all architectural proposals against the likelihood and scale of cost increase—and identifying and prioritizing those whose implementation clearly brings equivalent or larger business advantage—supports time-to-value and resource management.

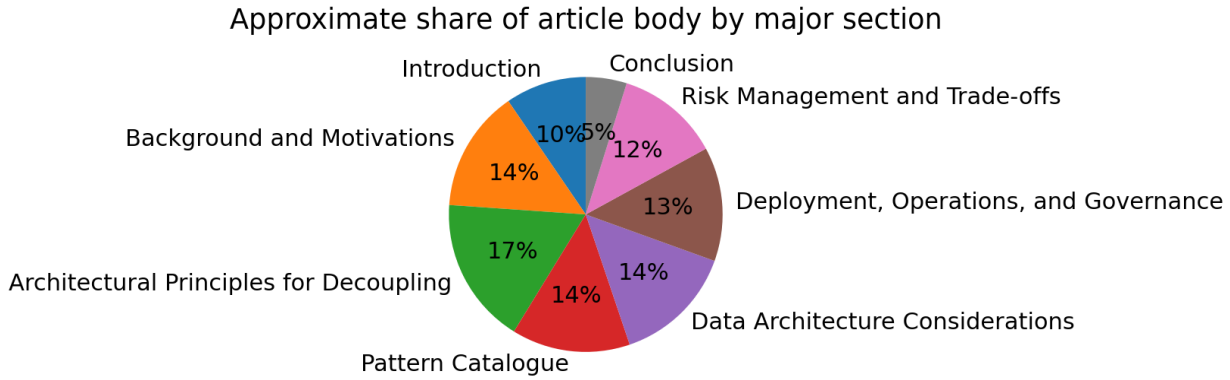
Discerning the respective timing of Real-Time and Sectioned Vanishing Point caveats above helps prioritize the sequence and targeting of initial decoupling investments, particularly when the closeness of a decision makes the influence of potential future cost cheaper to estimate than potential future cost worse. Formally adopting a decision framework into which all decoupling proposals are categorized and evaluated avoids the risk of expensive missteps: after all, both cost and complexity simply repay investment according to their own decoupling patterns, making initial patterns focus on replication of current business capabilities—in newly decoupled form.

8. Conclusion

Decoupling a tightly coupled monolith is a delicate operation akin to reducing the connections of an airplane mid-flight, yet with the right surgical patterns, the migration can be successful. Strangler fig and microservice refactoring provide geometric patterns commonly encountered; less familiar are the architectural implications of API gateways and Backends for Frontends. With an appropriate skill set, decoupling need not imply a shift from the traditional to the trendy; the strangle zoning pattern permits gradual change without fundamental redesign.

Health insurance providers would typically be crippled by the combination of an aging monolith, the non-discretionary nature of their work, and the demands of a best-of-breed Digital Enablement layer (provisioning entry and response points into and out of varied `as-is` internal processing). A slowdown in new regulatory demands and the availability of a skilled force able to implement change opened the way for alternatives that would gradually reduce the burden of development while rolling out products that delivered the easier-to-implement `Digital-Only` alternatives. In such circumstances, the demand for taming and modernizing the coupled monolith may grow even stronger.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17



8.1. Future Directions

A noted limitation is the analyzing of architectural patterns from a decoupling-systems perspective without an understanding of complete core-systems transformation, whose operation, strategies, and roadmaps are reflected in the design of operational and legacy-supporting systems. A more holistic layering can provide a generic reference architecture or architectural steeplechase, supporting step-by-step migration against principles of regulatory, data-governance, operational, and delivery-risk reduction.

Three observations consolidate the work. First, an explicit modularity-and-boundary definition expands upon the DDD Bounded Context hypothesis, decoupling implementation from analysis-stage strategic design. Second, a contrast between ownership of APIs in API Gateways and Backends-for-Frontends solutions clarifies presentation-layer integration and gatekeeping resourcing. Third, mapping market-facing systems addresses regulatory and risk-positioning concerns that define nonfunctional and non-capital-business–value trade-offs in prioritizing event-driven architecture.

9. References

1. Alwardt, B., Schaper, L. K., Marsh, C., & Rehman, S. (2021). FHIR as a foundation for interoperability in U.S. healthcare information exchange. Journal of the American Medical Informatics Association, 28(10), 2131–2139.
2. Assunção, W. K. G., Wolfart, D., da Silva, I. F., Domingos, D. C. P., Schmeing, E., Villaca, G. L. D., & Paza, D. N. (2021). Modernizing legacy systems with microservices: A roadmap. Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering, 149–159.
3. Amistapuram, K., Pandiri, L., Raju, V. R., Paleti, S., Singireddy, S., & Sheelam, G. K. (2025). AI-Based Cloud Infrastructure and MLOps Frameworks for Scalable Data

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

Engineering Across Banking and Insurance. In 2025 IEEE International Conference on Communication Networks and Computing (CNC) (pp. 186–192). IEEE. 2025 IEEE International Conference on Communication Networks and Computing (CNC). https://doi.org/10.1109/cnc68716.2025.11484532

4. Bass, L., Weber, I., & Zhu, L. (2022). DevOps: A software architect's perspective (2nd ed.). Addison-Wesley.
5. Beck, T., & Plattner, H. (2022). Event-driven architectures for enterprise digital transformation. IEEE Software, 39(5), 63–70.
6. Danghi, P. S., Maniraj, K., Jain, P., Adilakshmi, K., Garapati, R. S., & Jain, S. K. (2025). Artificial Intelligence Based Energy Optimization Framework for Wireless Sensor Networks. In 2025 IEEE 5th International Conference on ICT in Business Industry & Government (ICTBIG) (pp. 1–6). IEEE. 2025 IEEE 5th International Conference on ICT in Business Industry & Government (ICTBIG). https://doi.org/10.1109/ictbig68706.2025.11323860
7. Bellavista, P., Berrocal, J., Corradi, A., Das, S. K., Foschini, L., & Zanni, A. (2022). A survey on fog computing for the Internet of Things. Pervasive and Mobile Computing, 83, 101560.
8. Chen, M., Decary, M., & Zhang, Y. (2023). Cloud-native transformation for healthcare information systems. IEEE Access, 11, 45182–45197.
9. Kummari, D. N., Burugulla, J. K. R., Malempati, M., Amistapuram, K., Garapati, R. S., & Nagabhyru, K. C. (2025, December). Enhancing Audit Compliance and Operational Efficiency in Manufacturing and Commercial Insurance Through Agentic AI and Data Engineering Frameworks. In 2025 IEEE International Conference on Communication Networks and Computing (CNC) (pp. 714-720). IEEE.
10. Düllmann, T., & Koschel, A. (2021). Towards a microservices reference architecture for insurance companies. Proceedings of the International Conference on Advanced Service Computing.
11. Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2021). DevOps. IEEE Software, 38(2), 94–100.
12. Grieves, M., & Vickers, J. (2022). Digital twins for healthcare and insurance modernization. IEEE Computer, 55(4), 72–79.

13. Kolla, T. (2025). Anomaly Detection Models for Outlier Provider Behavior in Cost and Treatment Patterns. Journal of Computational Analysis & Applications, 34(12), 1204.
14. Haug, P. J., & Ferraro, J. P. (2022). Modern health information system architectures for interoperability. Journal of Healthcare Information Management, 36(2), 24–35.
15. IEEE Standards Association. (2022). IEEE standard for software architecture documentation. IEEE.
16. Khezri, H., & Afsarmanesh, H. (2023). API-driven modernization strategies for enterprise systems. Future Internet, 15(4), 126.
17. Kratzke, N., & Quint, P. C. (2021). Understanding cloud-native applications after 10 years of cloud computing. Journal of Systems and Software, 179, 111003.
18. Balamurugan, J., Bhuvaneswari, M., Aitha, A. R., Nagaraju, S., Viswanathan, R., & Dhasarathan, N. (2025, November). Explanatory Transformer-Based Sequential Recommendation: Assessing BERTRec with SASRec for Customer Behaviour Prediction. In 2025 International Conference on Emerging Engineering Technologies and Applications (IC-EETA) (pp. 470-475). IEEE.
19. Li, X., Zhang, J., & Wang, Y. (2024). Legacy application modernization using cloud-native microservices in healthcare enterprises. Journal of Systems Architecture, 147, 103021.
20. Newman, S. (2021). Building microservices (2nd ed.). O'Reilly Media.
21. Pahl, C., Jamshidi, P., & Zimmermann, O. (2022). Architectural principles for cloud-native software systems. IEEE Software, 39(3), 78–85.
22. Sudhakar, A. V. V., Inala, R., Verma, A. K., Nag, K., Pandey, V., & Anand, P. S. (2025, September). Hybrid Rule-Based and Machine Learning Framework for Embedding Anti-Discrimination Law in Automated Decision Systems. In 2025 International Conference on Intelligent Communication Networks and Computational Techniques (ICICNCT) (pp. 1-6). IEEE.
23. Richardson, C. (2021). Microservices patterns (2nd ed.). Manning Publications.
24. Taibi, D., Lenarduzzi, V., & Pahl, C. (2021). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 8(5), 20–29.
25. Radha, S., Gottimukkala, V. R. R., Thottara, S., & Vandhana, K. (2025, October). Adaptive Video Streaming Over 5G Networks Using Deep Reinforcement Learning with Closed-Loop

Feedback Mechanism for Bitrate Control. In 2025 International Conference on Communication, Computer, and Information Technology (IC3IT) (pp. 1-6). IEEE.

26. Zhang, Y., Chen, L., & Wang, H. (2023). Modernizing healthcare legacy systems through API-first and event-driven architectures. Journal of Systems and Software, 205, 111826.
27. Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2022). Microservices: The journey so far and challenges ahead. IEEE Software, 39(3), 24–35.
28. Kratzke, N., & Quint, P. C. (2021). Understanding cloud-native applications after 10 years of cloud computing—A systematic mapping study. Journal of Systems and Software, 179, 111003.
29. Kolla, S. K., & Bandi, V. D. V. K. (2025). Autonomous Clinical Monitoring Platforms Using Reinforcement Learning and Deep Neural Networks. International Journal of Advanced Research in Computer Science & Technology (IJARCST), 8(6), 13300-13313.
30. Lewis, J., & Fowler, M. (2021). Microservices architecture and evolutionary modernization. IEEE Software, 38(4), 68–75.
31. Li, X., Zhang, J., & Wang, Y. (2024). Legacy application modernization using cloud-native microservices in healthcare enterprises. Journal of Systems Architecture, 147, 103021.
32. Merson, P. (2021). Software architecture for developers (2nd ed.). Addison-Wesley.
33. Balamurugan, J., Aitha, A. R., Kishore, D. S. C., Reenaraj, T., Babu, T. S., & Kumar, B. B. (2025, November). Adaptive AI-Driven Optimization in Smart Manufacturing: A Hybrid Neural-Network and Genetic-Algorithm Approach. In 2025 International Conference on Emerging Engineering Technologies and Applications (IC-EETA) (pp. 690-697). IEEE.
34. Newman, S. (2021). Building microservices (2nd ed.). O'Reilly Media.
35. Pahl, C., Jamshidi, P., & Zimmermann, O. (2022). Architectural principles for cloud-native software systems. IEEE Software, 39(3), 78–85.
36. Richardson, C. (2021). Microservices patterns (2nd ed.). Manning Publications.
37. Srikanth, T., Segireddy, A. R., & Elavarasi, S. A. (2025, October). STaSFormer-SGAD: Semantic Triplet-Aware Spatial Flow-Guided Spatio-Temporal Graph for Anomaly Detection in Surveillance Videos. In 2025 International Conference on Communication, Computer, and Information Technology (IC3IT) (pp. 1-7). IEEE.

38. Taibi, D., Lenarduzzi, V., & Pahl, C. (2021). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 8(5), 20–29.
39. Wolfart, D., Assunção, W. K. G., da Silva, I. F., Domingos, D. C. P., Schmeing, E., Villaca, G. L. D., & Paza, D. N. (2021). Modernizing legacy systems with microservices: A roadmap. In Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (pp. 149–159).
40. Amistapuram, K., Pandiri, L., Raju, V. R., Paleti, S., Singireddy, S., & Sheelam, G. K. (2025, December). AI-Based Cloud Infrastructure and MLOps Frameworks for Scalable Data Engineering Across Banking and Insurance. In 2025 IEEE International Conference on Communication Networks and Computing (CNC) (pp. 186-192). IEEE.
41. Nogueira, V. L., Felizardo, F. S., Amaral, A. M. M., Assunção, W. K. G., & Colanzi, T. E. (2024). Insights on microservice architecture through the eyes of industry practitioners. arXiv.
42. Diggs, C., Doyle, M., Madan, A., Scott, S., Escamilla, E., Zimmer, J., Nekoo, N., Ursino, P., Bartholf, M., Robin, Z., Patel, A., Glasz, C., Macke, W., Kirk, P., Phillips, J., Sridharan, A., Wendt, D., Rosen, S., Naik, N., Brunelle, J. F., & Thaker, S. (2024). Leveraging LLMs for legacy code modernization: Challenges and opportunities for LLM-generated documentation. arXiv.
43. Gadi, A. L., Garapati, R. S., Inala, R., Singireddy, J., & Kapila, D. (2025, October). Robust Mutual Authentication for Distributed IoT Systems: Balancing Security and Efficiency. In International Conference on Microelectronics, Electromagnetics and Telecommunication (pp. 538-548). Cham: Springer Nature Switzerland.
44. Khezri, H., & Afsarmanesh, H. (2023). API-driven modernization strategies for enterprise systems. Future Internet, 15(4), 126.
45. Chen, M., Decary, M., & Zhang, Y. (2023). Cloud-native transformation for healthcare information systems. IEEE Access, 11, 45182–45197.
46. Beck, T., & Plattner, H. (2022). Event-driven architectures for enterprise digital transformation. IEEE Software, 39(5), 63–70.

P-ISSN: 3050-9556
E-ISSN: 3050-9564
EUROPEAN JOURNAL OF ANALYTICS AND ARTIFICIAL INTELLIGENCE
VOLUME: 03 ISSUE: 04
RECEIVED: OCTOBER 23 REVISED: NOVEMBER 07 ACCEPTED: NOVEMBER 28 PUBLISHED: DECEMBER 17

47. Oladosu, S. A., Ige, A. B., Ike, C. C., Adepoju, P. A., Amoo, O. O., & Afolabi, A. I. (2023). AI-driven security for next-generation data centers: Conceptualizing autonomous threat detection and response in cloud-connected environments. GSC Adv Res Rev, 15(2), 162-172.
48. Alwardt, B., Schaper, L. K., Marsh, C., & Rehman, S. (2021). FHIR as a foundation for interoperability in U.S. healthcare information exchange. Journal of the American Medical Informatics Association, 28(10), 2131–2139.
49. Haug, P. J., & Ferraro, J. P. (2022). Modern health information system architectures for interoperability. Journal of Healthcare Information Management, 36(2), 24–35.
50. Lebcir, I., Mageswari, S. U., Bhosale, Y. H., Nagubandi, A. R., & Mahabooba, M. M. Agile Strategic Management in the Age of Disruption: Leveraging AI and Data Analytics for Competitive Advantage.
51. Bellavista, P., Berrocal, J., Corradi, A., Das, S. K., Foschini, L., & Zanni, A. (2022). A survey on fog computing for the Internet of Things. Pervasive and Mobile Computing, 83, 101560.
52. Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2021). DevOps. IEEE Software, 38(2), 94–100.
53. IEEE Standards Association. (2022). IEEE standard for software architecture documentation. IEEE.
54. Inala, R., Kaulwar, P. K., Nagabhyru, K. C., Adusupalli, B., & Arun Raj, S. R. (2025, October). Leveraging IEC 61850 for Interoperable and Resilient Smart Grid Communication Architecture. In International Conference on Microelectronics, Electromagnetics and Telecommunication (pp. 549-566). Cham: Springer Nature Switzerland.
55. Nogueira, V. L., Felizardo, F. S., Amaral, A. M. M., Assunção, W. K. G., & Colanzi, T. E. (2024). Insights on microservice architecture through the eyes of industry practitioners. arXiv.
56. Diggs, C., Doyle, M., Madan, A., Scott, S., Escamilla, E., Zimmer, J., Nekoo, N., Ursino, P., Bartholf, M., Robin, Z., Patel, A., Glasz, C., Macke, W., Kirk, P., Phillips, J., Sridharan, A., Wendt, D., Rosen, S., Naik, N., Brunelle, J. F., & Thaker, S. (2024). Leveraging LLMs for legacy code modernization: Challenges and opportunities for LLM-generated documentation. arXiv.

57. Radhakrishnan, P., Manoranjithem, V., Garapati, R. S., Singh, S., & Praveen, R. V. S. (2025, November). Random Forest–XGBoost Hybrid Model for Early Detection of Breast Cancer in Medical Imaging Datasets. In 2025 IEEE 3rd Global Conference on Wireless Computing and Networking (GCWCN) (pp. 1-6). IEEE.
58. Khezri, H., & Afsarmanesh, H. (2023). API-driven modernization strategies for enterprise systems. Future Internet, 15(4), 126.
59. Beck, T., & Plattner, H. (2022). Event-driven architectures for enterprise digital transformation. IEEE Software, 39(5), 63–70.
60. Pareyani, S., Goswami, S., Geetha, Y., Dimri, S. K., Niharika, D. S., & Amistapuram, K. (2025, December). Smart Resource Allocation in Wireless Sensor Networks Through AI Techniques. In 2025 IEEE 5th International Conference on ICT in Business Industry & Government (ICTBIG) (pp. 1-6). IEEE.
61. Chen, M., Decary, M., & Zhang, Y. (2023). Cloud-native transformation for healthcare information systems. IEEE Access, 11, 45182–45197.
62. Alwardt, B., Schaper, L. K., Marsh, C., & Rehman, S. (2021). FHIR as a foundation for interoperability in U.S. healthcare information exchange. Journal of the American Medical Informatics Association, 28(10), 2131–2139.
63. Nigam, N., Sireesha, B., Ediga, P., Segireddy, A. R., & Bokde, S. (2025, December). Comparative Evaluation of Cloud Security Algorithms Using Multiple Classifiers with an Optimized Intrusion Detection System. In 2025 IEEE 5th International Conference on ICT in Business Industry & Government (ICTBIG) (pp. 1-6). IEEE.
64. Pahl, C., Jamshidi, P., & Zimmermann, O. (2022). Architectural principles for cloud-native software systems. IEEE Software, 39(3), 78–85.
65. Bass, L., Weber, I., & Zhu, L. (2022). DevOps: A Software Architect's Perspective (2nd ed.). Addison-Wesley.
66. Kolla, S. H., & Mangala, N. (2025). DESIGNING AUTONOMOUS LLM AGENT FRAMEWORKS USING GEN AI PIPELINES TO ENHANCE CUSTOMER SERVICE MANAGEMENT AND KNOWLEDGE WORKFLOWS. Lex Localis-Journal of Local Self-Government, 23 (S6), 9719–9733.
67. Richardson, C. (2021). Microservices Patterns (2nd ed.). Manning Publications.
68. Newman, S. (2021). Building Microservices (2nd ed.). O'Reilly Media.

69. Taibi, D., Lenarduzzi, V., & Pahl, C. (2021). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 8(5), 20–29.
70. Mangala, N., & Kolla, S. H. (2025). Real-Time Feature Engineering for Streaming AI Workloads Using PySpark and Azure Event Hubs. International Journal of Multidisciplinary Research in Science, Engineering, Technology & Management, 1(6), 93-105.
71. Wolfart, D., Assunção, W. K. G., da Silva, I. F., Domingos, D. C. P., Schmeing, E., Villaca, G. L. D., & Paza, D. N. (2021). Modernizing legacy systems with microservices: A roadmap. In Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (pp. 149–159).
72. IEEE Standards Association. (2022). IEEE Standard for Software Architecture Documentation. IEEE.
73. Haug, P. J., & Ferraro, J. P. (2022). Modern health information system architectures for interoperability. Journal of Healthcare Information Management, 36(2), 24–35.
74. Mattaparthi, R. (2025). GenAI-Augmented Diagnostic Reasoning for Diesel Engine Fault Triage: A Large Language Model Framework for Technician Decision Support at Scale. Journal of Material Sciences & Manufacturing Research, 6(12), 1.
75. Bellavista, P., Berrocal, J., Corradi, A., Das, S. K., Foschini, L., & Zanni, A. (2022). A survey on fog computing for the Internet of Things. Pervasive and Mobile Computing, 83, 101560.
76. Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2021). DevOps. IEEE Software, 38(2), 94–100.
77. Mangalampalli, B. M., Kolla, S. K., Bandi, V. D. V. K., Yandamuri, U. S., & Rani, P. S. (2025). Designing Intelligent Healthcare Ecosystems through Adaptive Data Integration and Autonomous Learning Systems. Vascular and Endovascular Review, 8(20s), 330-347.
78. Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2022). Microservices: The journey so far and challenges ahead. IEEE Software, 39(3), 24–35.
79. Li, X., Zhang, J., & Wang, Y. (2024). Legacy application modernization using cloud-native microservices in healthcare enterprises. Journal of Systems Architecture, 147, 103021.
80. Bandi, V. D. V. K. AI-Based Anomaly Detection Frameworks in Distributed Enterprise Data Systems.

81. Kratzke, N., & Quint, P. C. (2021). Understanding cloud-native applications after 10 years of cloud computing: A systematic mapping study. Journal of Systems and Software, 179, 111003.
82. Düllmann, T., & Koschel, A. (2021). Towards a microservices reference architecture for insurance companies. In Proceedings of the International Conference on Advanced Service Computing.
83. Davuluri, P. N. Integrating Artificial Intelligence into Event-Driven Financial Crime Compliance Platforms.
84. Assunção, W. K. G., Wolfart, D., Garcia, A., Colanzi, T. E., & Lucena, C. J. P. (2021). Are we speaking the industry language? The practice and literature of modernizing legacy systems with microservices. Proceedings of the 15th Brazilian Symposium on Software Components, Architectures and Reuse, 61–70.
85. Colanzi, T. E., Amaral, A. M. M., Assunção, W. K. G., Zavadski, A., Tanno, D., Garcia, A., & Lucena, C. J. P. (2021). Are we speaking the industry language? The practice and literature of modernizing legacy systems with microservices. Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse, 61–70.
86. Mangalampalli, B. M., & Kolla, S. K. (2025). Large Language Models for Automated Healthcare Data Dictionary Generation and Maintenance. Vascular and Endovascular Review, 8(20s), 363-375.
87. Hausotter, A., Koschel, A., Buchta, R., Grunewald, A., Lange, M., & Niemann, P. (2021). Towards a microservice reference architecture for insurance companies. Proceedings of the 13th International Conference on Advanced Service Computing, 5–9.
88. Hausotter, A., Koschel, A., Zuch, M., Busch, J., & Seewald, J. (2022). Microservices authentication and authorization from a German insurance perspective. International Journal on Advances in Security, 15(3–4), 65–74.
89. Diggs, C., Doyle, M., Madan, A., Scott, S., Escamilla, E., Zimmer, J., Nekoo, N., Ursino, P., Bartholf, M., Robin, Z., Patel, A., Glasz, C., Macke, W., Kirk, P., Phillips, J., Sridharan, A., Wendt, D., Rosen, S., Naik, N., Brunelle, J. F., & Thaker, S. (2024). Leveraging LLMs for legacy code modernization: Challenges and opportunities for LLM-generated documentation. arXiv.

90. Kolla, S. H. (2025). Autonomous Agentic Frameworks for Enterprise Service Operations and Intelligent Process Automation. International Journal of Science, Research and Technology, 8(4), 14643-14655.
91. Fávero, L. F., Almeida, N. R. de, & Affonso, F. J. (2025). A systematic mapping study on the modernization of legacy systems to microservice architecture. Applied System Innovation, 8(4), 86.
92. Kagga, S. R. (2025). Migrating legacy healthcare systems to cloud-native microservices with AI: Best practices and pitfalls. International Journal of Applied Mathematics, 38(2S), 914–949.
93. Nagubandi, A. R. (2025). FinTech and Financial Inclusion in Emerging Economies: An Empirical Assessment. Advances in Consumer Research.
94. Konda, L. P. (2025). Digital transformation in healthcare technology: Modernizing legacy expense management systems. International Journal of Computational and Experimental Science and Engineering.
95. Patel, R. (2025). API-first modernization and applied AI in insurance digital transformation: A multi-year enterprise case study. International Journal of Computational and Experimental Science and Engineering.
96. Davuluri, P. N. AI-Augmented Sanctions Screening: Enhancing Accuracy and Latency in Real Time Compliance Systems.
97. Dorairaj, A. (2025). From legacy to cloud: A case study in consolidating insurance producer compensation platforms. International Journal of Computational and Experimental Science and Engineering.
98. Kumar, V. (2025). Modernizing legacy enterprise systems: A framework for scalable insurance administration platforms. Journal of Information Systems Engineering and Management, 10(63s).
99. Enjam, G. R. (2024). Modernizing legacy insurance systems with microservices on Guidewire Cloud Platform. International Journal of Emerging Research in Engineering and Technology.
100. Bandi, V. D. V. K. (2024). Intelligent Data Platforms For Personalized Retail Analytics At Scale. Metallurgical and Materials Engineering, 30(4), 1011-1027.

101. Samson, F., Emmanuel, F. V., Kokala, A., Kalluri, K., et al. (2025). Modernising legacy insurance platforms: A microservices-driven approach. ResearchGate Preprint.
102. Seedat, M., Abbas, Q., & Ahmad, N. (2023). Systematic mapping of monolithic applications to microservices architecture. arXiv.
103. Mangala, N. (2025). Agentic Data Pipelines: Autonomous ELT Orchestration Using AI Agents on Microsoft Fabric and Databricks. International Journal of Computer Technology and Electronics Communication, 8(6), 11891-11907.
104. Wolfart, D., Assunção, W. K. G., da Silva, I. F., Domingos, D. C. P., Schmeing, E., Villaca, G. L. D., & Paza, D. N. (2021). Modernizing legacy systems with microservices: A roadmap. Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering, 149–159.
105. Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2022). Microservices: The journey so far and challenges ahead. IEEE Software, 39(3), 24–35.
106. Reddy, V. A. R. (2025). Journal of Rare Cardiovascular Diseases. Health, 5(3), 402-422.
107. Kratzke, N., & Quint, P. C. (2021). Understanding cloud-native applications after 10 years of cloud computing: A systematic mapping study. Journal of Systems and Software, 179, 111003.
108. Taibi, D., Lenarduzzi, V., & Pahl, C. (2021). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 8(5), 20–29.
109. Richardson, C. (2021). Microservices patterns (2nd ed.). Manning Publications.
110. Newman, S. (2021). Building microservices (2nd ed.). O'Reilly Media.
111. Kolla, T. (2025). Generative AI for Intelligent Medical Coding and Healthcare Analytics. International Journal of Advanced Research in Computer Science & Technology (IJARCST), 8(6), 13285-13299.