International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V74I8P104

Volume 74 Issue 8, 28-37, August 2026

Research Article

Domain-Driven Design: A Strategic Framework for Composable Commerce Architecture

Sethu Madhav Vure

Independent Researcher, Charlotte, NC, USA.

[1]*Corresponding Author : s.m.vure@gmail.com*

Received: 24 June 2026 Revised: 29 July 2026 Accepted: 17 August 2026 Published: 30 August 2026

Abstract *- Domain-Driven Design (henceforth referred to as DDD) is an industry-proven standard to handle large and complex monoliths and decompose them efficiently. This article analyzes the usefulness of DDD to perform this decomposition to align with the composable commerce principles laid down by MACH architecture. A retail platform currently in production for a couple of years that is built using these principles is evaluated to obtain statistical evidence in proving the efficacy of these best practices. The analysis utilized bounded contexts, ubiquitous language and context mapping – the three pillars of DDD and their role in achieving composability. A validation mechanism, "The Replace-It Test", is also proposed to evaluate boundary integrity. The findings in the article conclude that coupling impact is not defined by the platform selection but instead by the quality of boundary enforcement. It also indicates that strategic DDD is essential to begin building a composable system, while tactical DDD could be applied iteratively.*

Keywords *- Bounded-Contexts, Composable Architecture, Domain-Driven Design, Domain Events, Ubiquitous-Language.*

1. Introduction

Monolithic legacy commerce platforms are not modular functionally. This means they create multiple data source dependencies on modules to the point where it gets extremely costly to maintain. Microservices / composable architecture [3] is the suggested industry standard, especially in the commerce sector, enabling decoupling, autonomy [5] and faster rollouts. A system is considered modular when its components can be assessed, changed and replaced independently. Composable commerce supports applying that modularity to retail platforms. The methodology allows building components like Catalog, Pricing, Inventory and Order that can be analyzed, replaced, and extended without impacting the rest of the application [3]. However, there is a distinction between composability and modularity. A system can appear composable with separate services and deployments, but still may not be modular. Some of the causes could be a shared database or dependency on non-exposed functions. However, switching to MACH principles without a good plan introduces distributed monoliths, which are even harder to manage [7,8,9]. This is proven in Section 4 of this paper, where inventory and order domains are used to showcase deep coupling in monoliths that may not be solved in composable architectures unless a domain separation exists. When employing Domain-Driven Design, especially the strategic approach, it is possible to identify how to effectively draw the system boundaries that achieve the right level of decomposition. The key callout here is that domain modeling and clear bounded contexts are essential to be applied to MACH architectures to ensure they do not relegate to re-coupling. The comparison between monolithic and properly DDD applied solution systems for six metrics across two domains would clearly indicate this. The metrics include synchronous call coupling across two domains, change blast radius, regression impact, LOC updates per feature, level of data store isolation and number of issues after deployment.

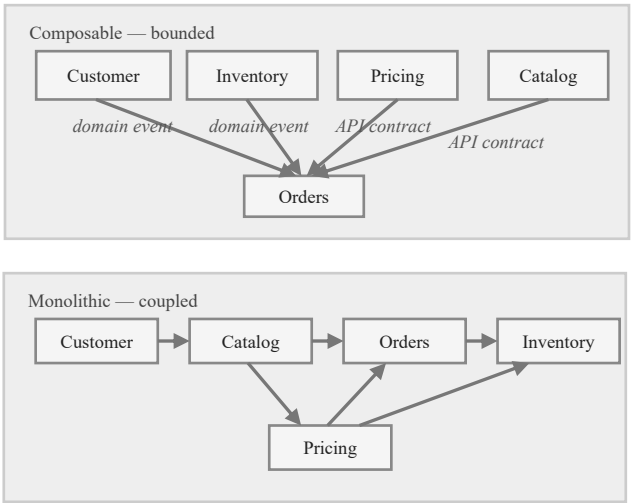


Fig. 1 Decomposition of domains in a composable commerce system. Each service adheres to a bounded context

2. Literature Overview

2.1. Domain-Driven Design

Eric Evans [2] proposed the concepts of Domain-Driven Design and, through this, coined bounded context as the means to clear boundary definitions. In a bounded context, elements, relationships and invariants are defined very clearly. He also suggested explicit translations when mediating between differing models. Vaughn Vernon [1] further added to this approach through large-scale structure patterns and strategic modeling. The key aspect to consider is to ensure the core domain is separated from generic and supporting sub-domains as a competitive differentiator.

2.2. Composable Architecture and MACH

The MACH Alliance [3] proposed a composable architecture as a blend of four architectural principles – Microservices, API-First, Cloud Native systems and Headless applications. This framework focuses on the technical aspects of an enterprise application. The design aspect of domain structures needs to be augmented by these definitions. Conway's Law [4] indicates that a system architecture is going to be shaped like the communication structure of the associated organization. The Team Topologies Model [5] puts a name to this: stream-aligned, enabling, complicated-subsystem, and platform teams, each mapped to a different point along the value stream.

2.3. Microservices Decomposition Patterns

Richardson [6] documents various patterns for service decomposition, some of which, like Decompose by Business Capability and Decompose by Subdomain, correlate closely to the strategic patterns of DDD. These patterns strongly callout ensuring clear boundary definitions of services as the foundation for a successful microservice architecture [7, 8, 9]. These boundaries are often drawn in alignment with existing technological or org structure convenience instead of clear domain definitions. This is causing tighter coupling across services, distributed transaction situations and shared data stores, which are typical indicators of distributed monoliths.

2.4. Empirical Evidence on Migration Outcomes

Taibi et al. [7] studied 21 organizations that migrated to microservice architectures and pointed out that incorrect service granularity and insufficient context boundary specifications are the major contributors to higher coupling situations after the migration.

Su, Li, and Taibi [8] reviewed documented reversal cases for organizations like Istio, Prime Video, Segment and InVision. As part of the research, they found that these cases are driven by cost, complexity, scalability and team misalignment. Faustino et al. [9] analyzed the migration effort from a quantitative viewpoint. The outcome indicated that redefining the system boundaries contributed to a large amount of rework cost.

Prior migration research largely used interview- and survey-based methods to identify causes for post-migration coupling. Taibi et al. [7] had the cause identified through practitioner report, while Faustino et al. [9] quantified the migration effort rather than boundary coupling. Lenarduzzi et al. [14] tracked code-level technical debt across a migration. The finding indicated that the debt growth slowed by ~90% with microservice architecture adoption. However, the composition of the debt shifted to maintainability-type issues. This result is broadly consistent with present findings, though it measures a different structure. Code-level debt describes the internal quality of a module. The metrics reported here describe coupling between modules. These two metrics need not move together. An additional alignment could be identified from Su, Li and Taibi [8]'s analysis of the documented reversals. These reversals from microservices to monolithic or modular deployments evaluated cost, complexity and organizational misalignment instead of re-emergence of coupling. This observation aligns with the study's central thesis, which puts boundary enforcement and not the platform choice as the operative variable. The present study extends this analysis in commerce and retail domains with direct measurements and confidence scores that are not called out in peer-reviewed migration collections.

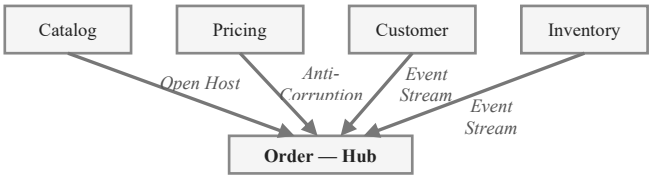


Fig. 2 Context map for a composable commerce platform. Shows relationship patterns between bounded contexts.

2.5. Research Gap

Domain-Driven Design's strategic patterns are well documented [1, 2]. Systematic reviews also exist to confirm that microservice identification is the most analyzed approach [11, 12]. In addition, practitioners have performed migration research that measured coupling impact as well as the technical debt for redefining boundaries [7]. Conformance metrics are also available to check the alignment of code to the decomposition patterns. The current article, however, proposes the boundary enforcement quality as an isolated variable independent of the platform choice.

Current published comparisons use architectural styles like monolith and microservices. Due to this approach, boundary quality is closely measured by the deployment topology. Evidence is also weighted towards interviews, surveys and grey literature. A first-hand static and repository analysis of migration impact inside one large organization is comparatively rare. Composable and MACH architectures are commercially active in the e-commerce and Retail sectors, but the peer-reviewed migration literature in this domain remains limited.

This study handles these gaps directly through below three research questions:

RQ1 – how do coupling indicators that are directly measurable differ between a nominally bounded legacy commerce system and a new system where bounded contexts are architecturally enforced?

RQ2 – Which observed differences of the analyses can be attributed to specific DDD patterns and through what mechanism?

RQ3 – Does enforcing boundary contexts account for these differences, independent of the platforms?

3. Materials and Methods

The article correlates the conceptual framework prescribed in the subsections 3.1 to 3.5 to the quantitative analysis indicated in Section 4. The concepts apply industry-standard DDD patterns to the commerce domain. Measurements captured in the quantitative analysis are from the production systems of a large retail organization that migrated from the legacy monolith (the *Nominal* system) to the domain-driven solution built on a composable commerce architecture (the *Enforced* system).

The Inventory domain was selected for the comparison, as it has the most complete instrumentation across both systems. It also has the sharpest structural contrast across the boundaries. It also offers a well-documented detail of post-deployment operational status. Thus, analytically, this is the most productive domain for comparison.

Legacy system analysis included static validation of three independent sets of evidence.

(a) Pipeline execution configuration files – these hold the component correlation to cross-domain dependencies,
(b) Data repository ORM files and the corresponding data objects (24 files across multi-module configuration layers and 130 tables associated with the repositories) – these provide info about the data ownership across boundaries and
(c) The source code and the 219 feature commits associated with the packages that capture the business logic extensions.

These three artefacts are holistic evidence that operate independently and cannot create empirical fabrications, thereby resulting in a strong quantitative confirmation of the proposed concepts.

MACH system analysis included (a) analyzing the git commit history (of 124 commits) of the features that span both Inventory Service and Order Orchestrator, (b) static analysis of dependency injection patterns that indicate over the boundary dependencies per user journeys and (c) datasource configuration files that confirm the adherence of data store per service principle.

3.1. Bounded Context Identification

Bounded contexts are identified through the terms of the context and their meanings. The meaning should not create ambiguity within a team's working vocabulary. Consider Product as a term, for example. It could be interpreted differently based on the context: a content element with media and taxonomy attributes in the Catalog context, a price-book entry with SKU and cost attributes in the Pricing context and a line item with the tax and fulfillment option info in the Checkout context. Each should be a distinct model that should not be merged.

Event Storming [10] could be used for the identification of the contexts and boundaries. The approach suggests that events are grouped based on the model they are associated with, and the groups are bounded by requirements to perform translation. If multiple contexts claim ownership for the same event, it indicates there is a boundary ambiguity that would require further resolution.

3.2. Ubiquitous Language

It is a best practice to build a glossary of the terms associated with the models. This could also be identified through Event Storming. The approach involves collaborative sessions between domain experts and engineers. Once identified and aligned, care should be taken that this glossary is followed diligently in realization.

For example, the terms should be used in code through class and API resource names, as well as the method signatures. Ubiquitous language is the process of achieving this consistency through a glossary built up. If the common terms are identified across contexts, context maps should be used instead of a global canonical model.

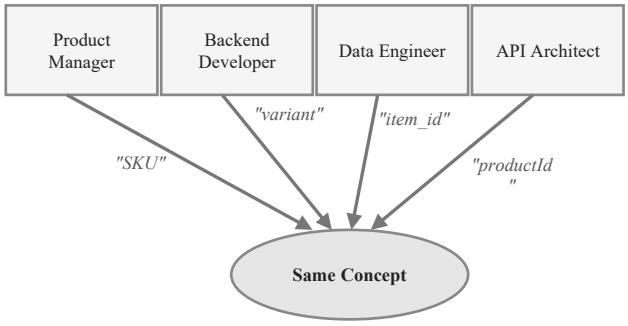


Fig. 3 Ubiquitous language enables consolidation of role-specific naming like SKU, variant, item_id, and productId to a single shared concept

3.3. Context Mapping

DDD pattern vocabulary could be used to classify context maps. The table (table 1) summarizes the patterns applied in the architecture under focus.

Table 1. Context mapping patterns in the case architecture.

Relationship	Coupling	Example
Shared Kernel	High	Catalog ↔ Search (shared product projection)
Customer/Supplier	Medium	Pricing → Checkout (price API contract)
Conformist	Medium	Promotions → Catalog (follows catalog model)
Anti-Corruption Layer	Low	Legacy OMS → Checkout (translation layer)
Open Host Service	Low	Catalog → Storefront (public content API)
Published Language	Low	All contexts → Analytics (event schema)

3.4. Aggregate Design

Within each bounded context, consistent boundaries could be established by defining the aggregates. These are the set of objects that should be transactionally consistent. Each aggregate is also a single root entity that controls external access. For example, `Order`, as the aggregate for the `Checkout` context, includes `OrderLine`, `ShippingAddress` and `PaymentInstruction`. `Inventory` is an aggregate in a separate context because inventory reservation is an eventually consistent operation that should not block order capture.

3.5. Domain Events

When the state of domain objects is updated, they emit domain events. These events are usually consumed by other contexts. The naming and structure of the events would strictly follow the same rules of ubiquitous language that govern the owner context. `OrderPlaced`, `PriceListPublished` and `InventoryReserved` are a few examples. When other contexts process these events, the processing usually happens through an Anti-Corruption Layer. This layer translates the event structure from the publisher to the consumer schema. This prevents the publisher's model changes from propagating across the boundary.

4. Results and Discussion

4.1. The Distributed Monolith Pattern

When the legacy platform's order module is analyzed functionally, three structural indicators of a distributed monolith were uncovered in association with the module of choice – Inventory.

Firstly, a dedicated inventory package (com.company.inventory) was identified within the codebase. However, the elements within were simply HTTP/SOAP adapters to a centralized inventory management system. They are not functioning independently, even from a capability standpoint. The six processors that perform the inventory operations are part of the order processing namespace pipeline (com.company.order.processor). These processors get executed sequentially as part of the order execution process in the commerce checkout module. For a proper decoupled architecture, inventory operations should be in their own bounded context instead of a simple client library used by the Order domain.

Second, the inventory reservation state is stored in the order repository tables. The order pipeline components persist the state details directly to the tables of the associated order module. A change in inventory structure penetrates all the way to order execution functionality, indicating high coupling.

Third, the Order module owns eleven tables storing inventory, loyalty and promotion data when it should manage none of these contexts. Together, all these indicators confirm the distributed monolith state of the legacy system: Inventory as a named context is owned, controlled and persisted by Order domain [7, 8].

4.2. Impact of Boundary Enforcement

When bounded contexts are enforced at the architecture stage in a MACH-compliant system, significant structural differences were observed across all six dimensions under focus. Table 2 captures the metrics from the analysis of the two systems. The Impact column indicates the result as well as the confidence score (H = HIGH, M = MEDIUM) of the analysis.

Table 2. Nominal vs enforced bounded context: directly measured comparison

Metric	Nominal *(legacy)*	Enforced *(MACH)*	Impact (conf.)
Sync calls — PDP	3 in-process	5 HTTP *(explicit)*	Coupling made explicit (H)
Sync calls — Checkout	11 pipeline stages (in-JVM)	6 HTTP; *inv. async*	Inventory decoupled (H)
LOC/feature — median	~325 *(n=219 commits)*	13 *(n=124 commits)*	~25× (H)
Regression — change scope	4 modules	1 module *(median)*	4× (M/H)
Regression — QA blast radius	7 functional areas	1–2 services	3.5–7× (M/H)
Cross-domain data coupling	11 tables	0 tables	11→0 (H)
Incidents / 100 deploys	467 *(3 releases)*	18 *(GKE revisions)*	~26× (M)

4.2.1. Measurement Notes

Analysis of pipeline configurations and the component dependencies defined in configurations is used to identify the Sync call counts. The legacy system's process invocations are like the inter-service calls. However, the key difference is that they are not visible at runtime and create additional complexities to be versioned and circuit breakable. In the new system, inventory reservation at checkout is asynchronous (albeit introducing eventual consistency issues). In legacy, it is synchronous and baked into the order confirmation process. Analysis of git commits for both legacy and new systems helped identify the LOC and regression scope impact numbers. (Legacy commit count is 219, spanning Inventory, Checkout and Order modules. New system commit count is 124, spanning Inventory and Order services. A deep repository analysis of 24 ORM configuration files in focus was conducted to understand the data coupling. Post-release status for legacy systems for three consecutive releases in a 90-day window was compared to the new system's counterparts from GKE deployment history to understand the correlation of change radius to the issues created or identified.

4.3. Pattern Effectiveness

Table 2 reports six directly measured differences. Table 3 attributes each of them to a specific pattern and adds the mechanism through which the pattern contributed.

Table 3. Metric to DDD Pattern Mapping with Mechanism.

DDD Pattern / Metric	Nominal / Enforced	Mechanism	Conf Score
Bounded Context – Data Coupling	11 / 0 tables	Cross-context access made inexpressible	H
Bounded Context – Regression scope	4 / 1 modules	No shared storage for propagation	M/H
Ubiquitous Language – LOC / Feature median	325 / 13	Removes cross-module translation	M
ACL – QA blast radius	7 areas / 1-2 services	Schema change impact minimal	M/H
Context Map – Sync call profile	3 in-proc + 11 stages / 5 explicit + 6 async http	Declared contracts replacing implicit calls	H
Domain Events – Incidents / 100 deploys	467 / 18	Removes synchronous failure propagation	M

4.3.1. Bounded Context

The problem of accessing cross-context tables from a nominal system is made harder from an implementation viewpoint in the enforced system, rather than merely discouraged architecturally [15, 16]. This is what is being called out as enforcement in the current paper. Having no shared tables does not indicate minimal or no coupling. Event contracts discussed in Domain Events below would open the scope to introduce dependencies.

4.3.2. Ubiquitous Language

Reduction of LOC by ~25-fold cannot be attributed only to vocabulary consistency. Other factors like platform approaches, frameworks, managed persistence, and granularity conventions also play a role. This is also reflected in the confidence score, which is set to MEDIUM.

4.3.3. Anti-Corruption Layer

ACL ensures that contract changes have minimal impact on the translation layer. However, it is not free; it adds code at the boundary. This results in a trade-off between blast radius and change impact control.

4.3.4. Context Map

Dependency declarations could be made versioned, circuit-breakable and rate-limited. This is hard to achieve in a nominal system. There could be increased inter-service calls in the enforced system compared to method invocations in legacy. However, they indicate lower effective coupling than ungoverned legacy invocations [17]. Typed relationships at checkout helped achieve synchronous vs event-based integration demarcations. This is the key reason why the Replace-It Test mentioned below succeeds.

4.3.5. Domain Events

Nominal system implemented inventory reservation as a synchronous block within order confirmation. Because of this, a fault in inventory reservation results in order failure. This can be resolved through event choreography by delaying the reservation instead of failing the order. This also reduces the modules affected per release, which would, in addition, ensure a reduction in the areas exposed to defects. Confidence score is set to MEDIUM as the comparison of both the applications involves different measurement windows and incident reporting systems.

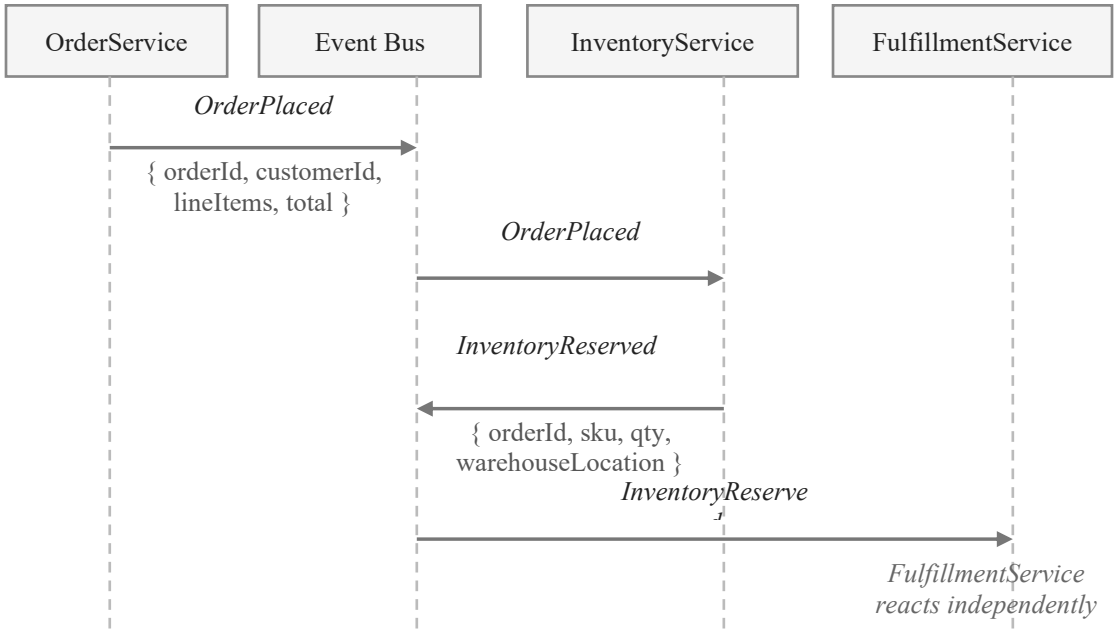


Fig. 4 Sequence diagram depicting order placement and inventory reservation connected as domain events with OrderService, the event bus, InventoryService, and FulfillmentService

4.3.6. Confounders

The two systems have differences with respect to platform, frameworks, teams, tooling and period of build. Boundary enforcement is not the only variable that is changed.

4.4. The Replace-It Test

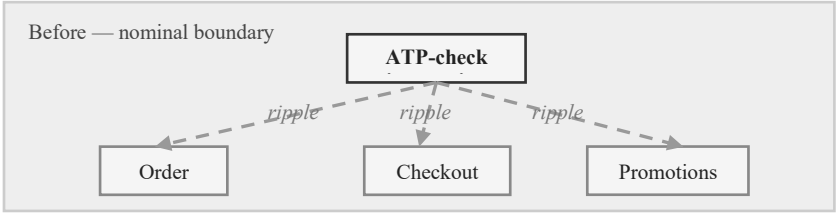
The "Replace-It Test" is a design-level validation tool to test coupling impact. The test would try substituting a service of a bounded context with a different vendor SaaS, an in-house build, or a third-party package. If the change is possible without modification of the consuming contexts beyond their Anti-Corruption layer, the bounded context is considered well defined.

The legacy inventory scenario failed this test. To substitute the external ATP-check integration in the legacy platform, each of the following steps was required.

- Identify the six pipeline stages in the order processor namespace that use the integration.
- Get an understanding of the checkout pipeline's operation sequence.
- Modify the Order data repository to update the inventory reservation record and
- Regression test the Order, Inventory, and Promotions domains that are associated with the checkout pipeline.

Three contextual concerns needed to be updated to change one external service configuration. This clearly indicates the cascading blast radius of regression testing of a nominal boundary that exists only in package naming.

The MACH Inventory service passes the same test. When the upstream OMS adapter (connecting to OMS, which owns the inventory master) is substituted, the changes are limited only to that deployable unit and its published event contracts. No downstream service is impacted due to this change. This is because inventory reservation is implemented via asynchronous events. A checkout never synchronously invokes the Inventory service. Thus, the Replace-It Test takes care of two claims here. It checks both MACH's vendor-replaceability and DDD's context isolation principle.



single configuration change propagates into three contextual concerns

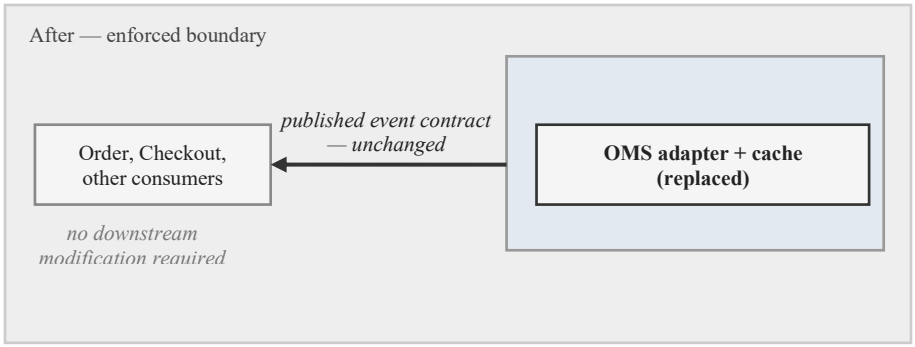


Fig. 5 Replace-It Test substitution impact. Before: replacing Inventory ATP check impacts Order, Checkout, and Promotions modules. After: the same substitution is limited to an API contract change

The comparison above uses two variables for nominal and enforced systems: platform and boundary enforcement. Table 4 maps the two variables to a four-scenario matrix, with scenarios labeled from S1 to S4. If boundary enforcement is used as an operative variable, S2 and S4 should behave similarly, and so should S1 and S3. This is because behavior is the right measure of enforcement. The data support S1 and S2 directly. Analysis of S3 and S4 is derived from evidence.

Though S1 is a monolith, Section 4.1 already shows com.company.The inventory package was an accurate boundary definition. However, this scenario still failed the test because its processors are associated with the order processing pipeline instead. Thus, S1 provides direct evidence of the result of the boundary label without enforcement.

Table 4. Four-Scenario Matrix mapping platform and boundary enforcement are separately

Scenario	Platform	Boundary Enforcement	Status
S1	Monolith (legacy)	Nominal (package naming only)	Observed
S2	MACH / Composable	Enforced (structural)	Observed
S3	MACH / Composable	Nominal	Counterfactual
S4	Modular monolith	Enforced	Counterfactual

Table 5. Reasoning of metrics for S3 (MACH without enforced boundaries) along with the mechanism

Metric	S2 (observed)	S3 (predicted)	Basis
Data coupling	0 tables	≥0, plausibly ~11	The platform alone does not prevent a shared datastore. It is a catalogued microservice anti-pattern [15]
Sync calls (checkout)	6 HTTP + async	≥11, now networked	Simply lifting and shifting results in the same call sequence. Remote calls play the same role as method invocations in legacy.
Regression scope	1 module	≥4 deployables	Data and contract dependencies create change impacts, not process boundaries.
Replace-It Test	Pass (L3)	Fail, more expensively than S1	Releasing the substitution requires coordination across services.

Building a distributed system without boundary enforcement increases the cost of coupling instead of avoiding it. This is because the dependencies will cross over a network and a release boundary.

S4 is a more interesting subject because the evidence is not straightforward. Faustino et al. [9] identified migration effort and performance separately, that is, first modularizing the monolith and then distributing it to services.

Most of the costs were associated to modularizing step. If code-level criteria alone are considered, S4 passes. However, Su, Li, and Taibi's review [8] found that the reversal of modular monoliths usually causes shared databases. This is one criterion that S4 would fail.

Per metric, S4's data coupling would not reach zero. This is because shared schema with module-level access is a convention and not an enforcement. Regression scope and

blast radius would land close to S2, because a build system can enforce code-level module boundaries. In this case, the Replace-It Test would score L2 (refer to Table 6). This is because a substitution will limit the change within the owner module, but would still need a coordinated release.

This analysis divides the boundary enforcement into two unequal parts. Boundary enforcement for behavior in a monolith can be achieved through tooling and conventions. However, data boundaries would need physical separation to be real. Thus, the legacy system failed both. The modular monolith passes first, but not reliably second. MACH passes both.

4.4.1. Falsifiability

The claim is falsifiable in either situation. A MACH system that conforms to enforced boundary principles but still has a significant blast radius would not agree to the claim that boundary enforcement is the operative variable. On the other hand, a modular monolith that passes the Replace-It Test would support it.

Table 6 enhances the Replace-It Test from a narrated claim to a scoring system. This enhances the test's reusability in similar studies.

Table 6. Replace-It test scoring rubric

Level	Criterion
L0 – Not bounded	Substitution results in changes to the consuming contexts and shared persistence layer.
L1 – Nominally bounded	Substitution is limited through naming conventions. More than one deployable or shared schema would require a change.
L2 – Contract bounded	Substitution is limited to the owner module and its ACL. Release requires consumer coordination.
L3 – Fully bounded	Substitution is limited to the owner module and its ACL. Release would require no changes to consumers or coordination.

When scored against this rubric, the legacy system achieved L0/L1 level based on evidence in Section 4.4 above, as the substitution touched consumer contexts and shared tables. MACH inventory context, on the other hand, was able to achieve L3 as no downstream changes or release coordination were required. The approach has real limits, however. It works at design time but not at runtime. The substitution here was reasoned through instead of being carried out. It was also scored by one assessor on one domain. Checking that agreement across assessors and domains is left for future work.

4.5. Strategic vs Tactical DDD

Composable architecture usually begins with service decomposition. Care should be taken that Strategic DDD approaches to define the bounded contexts, ubiquitous language and context maps are completed before the service decomposition. Tactical DDD, which takes care of defining aggregates, entities, repositories and factories, could be held off till a complex situation is encountered, which would demand in-depth modeling. CRUD-based domain services and SaaS supported functionalities could be envisioned without tactical DDD in straightforward cases.

4.6. Evidence Integrity

The case rests on three measurements, which are independently measurable to track the scope definition.

Pipeline configurations deal with component interdependencies. Data repository definitions tackle the database-level couplings. Package analysis reveals code-level invocations. An update to one measurement would not fabricate the other categories, thereby providing strong evidential support.

Data coupling resulting in a [11 → 0, HIGH] comparative metric is the highest confidence finding and is quite unambiguous. This is because of the exhaustive inspection of the 24 repository mappings and 3 data source configurations.

Git derived metrics to compute the LOC change per feature. This generated a [~325 → 13, M/H] metric that uses a large enough sample (219 legacy / 124 new system commits) whose median cannot be distorted by individual code contributions. While these two metrics themselves provide significantly strong directional conformity, coupling these with feature-specific regression impact strengthens the case further. This analysis is based on the regression reports owned and managed by the operations support team.

4.7. Metric Selection

The metrics in Table 2 are derived from the artefacts associated with both systems. They do not use the formal structural coupling and conformance metrics of microservices patterns. Structural coupling metrics such as Panichella, Rahman, and Taibi's [18] require runtime instrumentation, which is available only on the enforced systems. On the other hand, automated conformance checks against decomposition patterns [13] could be used to check whether the code follows the architecture. The Replace-It test in Section 4.4 instead checks whether the architecture handles a dependency substitution, which is a more design-time question. The two approaches are complementary rather than competing.

The decomposition criteria proposed by Gysel et al. [19] are applied ahead of implementation time and derive service boundaries from sixteen coupling criteria. The present study inverts the order. Boundaries are already in production, and the six metrics validate them post hoc against a working

system. Derivation and validation are different problems associated with coupling; this study addresses the latter.

5. Conclusion

The article analyzed and validated that the concept of Domain-Driven Design is a foundational process for composable architecture, especially in the commerce domains. The most important finding is that MACH architecture is indeed central to composability, but it is not enough to control the re-emergence of distributed monolith symptoms. Service boundaries should be enforced through bounded contexts. They should be supported by ubiquitous language and contextual mappings.

These DDD components, a.k.a bounded contexts, ubiquitous language and context maps were proven to be instrumental to high cohesion across services through simpler versioning and circuit breaking. “Replace-it” test provides a well-grounded means to validate boundary conformity.

This analytical approach could be further extended to commerce verticals across the enterprise to analyze the impact of Domain-Driven Design on cross-system synchrony.

5.1. Ethical Considerations

The study has used internal engineering artifacts of a production commerce platform. Examples of such artefacts are pipeline configurations, source code, data repository mappings and deployment history. Any customer data, personally identifiable information or confidential business metrics were not used or reported. The organization is not identified in this paper. All findings are presented at an architectural pattern and metric level instead of at the implementation detail.

Conflicts of Interest

The author declares that there is no conflict of interest regarding the publication of this paper.

Footnotes

[1]MACH: Microservices, API-first, Cloud-native SaaS, Headless. The MACH Alliance (2020) defines composable architecture along these four axes. See https://machalliance.org.

References

[1] Vaughn Vernon, *Implementing Domain-Driven Design*, Addison-Wesley Professional, 2013. [Google Scholar]

[2] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, Pearson, 2003. [Publisher Link]

[3] MACH Alliance, MACH Architecture Principles. [Online]. Available: https://machalliance.org/

[4] Melvin E. Conway, “How do Committees Invent?,” *Datamation*, vol. 14, no. 4, pp. 28-31, 1968. [Google Scholar]

[5] Matthew Skelton, and Manuel Pais, *Team Topologies*, 2nd ed., IT Revolution Press, 2019. [Publisher Link]

[6] Chris Richardson, *Microservices Patterns*, Manning Publications, 2018. [Publisher Link]

[7] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl, “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22-32, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[8] Ruoyu Su, Xiaozhou Li, and Davide Taibi, “From Microservice to Monolith: A Multivocal Literature Review,” *Electronics*, vol. 13, no. 8, pp. 1-16, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[9] Diogo Faustino et al., “Stepwise Migration of a Monolith to a Microservices Architecture: Performance and Migration Effort Evaluation,” *Performance Evaluation*, vol. 164, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[10] Alberto Brandolini, Introducing EventStorming: An Act of Deliberate Collective Sensemaking, Leanpub, 2021. [Online]. Available: https://www.eventstorming.com/book/

[11] Chenxing Zhong et al., “Domain-Driven Design for Microservices: An Evidence based Investigation,” *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1425-1449, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[12] J. Sangabriel-Alarcón et al., “Domain-Driven Design in Microservices-based Systems Development: A Systematic Literature Review and Thematic Analysis,” *Programming and Computer Software*, vol. 50, no. 8, pp. 742-770, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[13] Uwe Zdun, Elena Navarro, and Frank Leymann, “Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns,” *Service-Oriented Computing: 15th International Conference, ICSOC 2017*, Malaga, Spain, vol. 10601, pp. 411-429, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[14] Valentina Lenarduzzi et al., “Does Migrating a Monolithic System to Microservices Decrease the Technical Debt?,” *Journal of Systems and Software*, vol. 169, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[15] Davide Taibi, and Valentina Lenarduzzi, “On the Definition of Microservice Bad Smells,” *IEEE Software*, vol. 35, no. 3, pp. 56-62, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[16] D.L. Parnas, “On the Criteria to be used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972. [CrossRef] [Google Scholar] [Publisher Link]

[17] Stefan Kapferer, and Olaf Zimmermann, "Domain-Driven Service Design: Context Modeling, Model Refactoring and Contract Generation," *Service-Oriented Computing: 14th Symposium and Summer School on Service-Oriented Computing, SummerSOC 2020*, Crete, Greece, vol. 1310, pp. 189-208, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[18] Sebastiano Panichella, Mohammad Imranur Rahman, and Davide Taibi, "Structural Coupling for Microservices," *Proceedings of the 11th International Conference on Cloud Computing and Services Science CLOSER*, vol. 1, pp. 280-287, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[19] Michale Gysel et al., "Service Cutter: A Systematic Approach to Service Decomposition," *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016*, Vienna, Austria, vol. 9846, pp. 185-200, 2016. [CrossRef] [Google Scholar] [Publisher Link]