RESEARCH ARTICLE

Domain-Driven Design for Decomposing Legacy .NET Applications into Microservices: Effects on Service Coupling and Maintainability

Sunny Hundwani
Web Application Architect, Bloomberg Industry Group, USA
Email: Hundwani.career@gmail.com

Received	Revised	Accepted	Published
11 May 2024	13 June 2024	1 August 2024	15 September 2024

Abstract

Background: Decomposing a legacy application into microservices is fundamentally a boundary-design problem. Domain-Driven Design (DDD) offers bounded contexts, subdomains, ubiquitous language, and explicit ownership as mechanisms for discovering business-aligned boundaries, but a domain-correct cut can still create excessive distributed coupling if code, data, runtime, or transaction dependencies cross those boundaries.

Objective: This systematic review synthesised evidence published through 31 December 2021 on DDD-informed and closely related monolith-decomposition approaches, with emphasis on service coupling, cohesion, maintainability, and transferability to legacy .NET systems.

Methods: Structured searches covered major software-engineering publishers and indexes and were supplemented by citation chaining. Peer-reviewed empirical studies were eligible when they evaluated legacy/monolith decomposition, service identification, or DDD-informed boundary design and reported coupling, cohesion, modularity, maintainability, evolvability, or migration outcomes. Fifty-eight unique records were screened, 29 full texts were assessed, and 17 primary studies were included.

Results: DDD was most useful for proposing semantically coherent service boundaries, particularly when bounded contexts were combined with business capabilities or domain models. Across empirical studies, stronger decomposition results were obtained when semantic boundaries were validated against static dependencies, runtime traces, or migration constraints. Industrial cases reported improved maintainability, development speed, or modularity when bounded contexts and incremental facades were used, but shared data, transaction coupling, cyclic calls, and over-fragmentation remained major risks. Direct peer-reviewed evidence for legacy .NET decomposition was sparse through 2021; contemporaneous Microsoft .NET guidance nevertheless provided a platform-specific implementation model consistent with the architecture-independent findings.

Conclusion: DDD should be treated as a semantic boundary-discovery method rather than a stand-alone decomposition algorithm. For legacy .NET systems, the most defensible approach is to identify bounded contexts, validate them against source/runtime/data dependencies, enforce ownership inside a modularised codebase, and extract only boundaries that demonstrate low change coupling and operational autonomy.

Keywords: domain-driven design; bounded context; legacy modernisation; .NET; microservices; service decomposition; coupling; cohesion; maintainability; systematic review

How to cite: Hundwani S. Domain-driven design for decomposing legacy .NET applications into microservices: effects on service coupling and maintainability. Int J Bus Comput Sci. 2024;22–32.

1. Introduction

Legacy enterprise applications frequently accumulate architectural dependencies that no longer reflect the business structure that originally motivated the system. A feature change that appears local may require edits across presentation, service, persistence, and integration layers, followed by a complete rebuild and coordinated deployment. Classical software-engineering research has long treated modularity as a mechanism for controlling change: modules should hide design decisions, exhibit strong internal cohesion, and minimise external coupling [6-8]. Microservices pursue similar objectives at a distributed level, but they make poor boundaries more expensive because a misplaced dependency becomes a network call, a contract, an additional deployment concern, or a distributed transaction [9-15,39,40,42]. Consequently, the central migration question is not whether a legacy application can be split, but where its responsibilities should be separated so that most future change remains local.

Domain-Driven Design (DDD) provides a business-centred answer to that problem. Strategic DDD divides a complex domain into subdomains and bounded contexts, each with an internally consistent model and ubiquitous language [4,5]. A bounded context does not automatically equal one physical microservice, but it defines an ownership boundary within which domain concepts can evolve consistently. This is attractive for microservice decomposition because the boundary is derived from business meaning rather than from accidental package structures or historical layers. Service Cutter and later DDD-oriented modelling work operationalised these ideas by combining domain entities, aggregates, relationships, and context maps with explicit decomposition criteria [22,31-35].

The relationship between DDD and coupling, however, is conditional. A bounded context may be conceptually coherent while depending on shared database tables, global transactions, common utility libraries, or synchronous workflows that cross several contexts. If those dependencies are left unchanged, migration can transform compile-time coupling into remote coupling without improving maintainability. Empirical decomposition research therefore increasingly combines business/domain information with static source-code analysis, runtime traces, data-flow relationships, topic models, graph clustering, or learned code representations [21,23-30]. These approaches offer a way to

test whether a proposed domain boundary corresponds to actual dependency structure.

Maintainability is a major motivation for monolith-to-microservices migration. Industry studies have reported legacy complexity, release coordination, and scalability as frequent drivers, while also identifying service-cut design as one of the most difficult migration decisions [16-21]. The benefits are not guaranteed. Microservices introduce distributed tracing, deployment automation, contract evolution, monitoring, and failure handling, while excessively fine-grained services can increase cognitive and operational overhead [12-15,36]. Thus, maintainability should be interpreted as change isolation and comprehensibility across the whole system, not merely as smaller code repositories.

The .NET context adds an important platform question. By 2021, Microsoft guidance for containerised .NET applications explicitly described DDD-oriented microservices, bounded contexts, aggregate boundaries, per-service persistence, and eShopOnContainers as a reference implementation [37,38]. Nevertheless, peer-reviewed empirical studies that specifically decomposed legacy .NET monoliths using DDD were scarce within the evidence window through 2021. The architectural mechanisms under review, semantic cohesion, dependency control, data ownership, and independent change, are largely language-independent, but automated tooling and code-analysis results may not transfer directly from Java-heavy research to C#/.NET estates.

This review therefore addressed four questions. RQ1: How was DDD or business-domain knowledge used to identify candidate microservice boundaries through 2021? RQ2: What effects were reported on coupling, cohesion, and dependency structure? RQ3: What evidence linked decomposition to maintainability, evolvability, or migration effort? RQ4: What conclusions can reasonably be transferred to legacy .NET applications? The synthesis separates semantic boundary discovery from technical boundary validation and treats microservice extraction as a later decision rather than an automatic consequence of drawing a context map.

2. Methods

The review followed systematic-review principles used in software engineering and PRISMA 2020 reporting concepts [1-3]. The evidence window was restricted to publications available up to 31 December 2021 to establish a fixed historical evidence horizon. Searches combined terms for domain-driven design, bounded context, business capability, monolith, legacy, microservice, decomposition, migration, extraction, refactoring, coupling, cohesion, modularity, maintainability, and evolvability. Major publisher/index sources included IEEE Xplore, ACM Digital Library, ScienceDirect, SpringerLink, Wiley, and peer-reviewed conference repositories. Backward and forward citation chaining was used for highly relevant primary studies and reviews available through 2021.

Studies were eligible when they evaluated an actual legacy/monolithic system, a reproducible decomposition method, a DDD modelling/decomposition technique with validation, or industrial microservice migration practice and reported outcomes that informed coupling, cohesion, maintainability, evolvability, migration complexity, or service-boundary quality. Both journal articles and peer-reviewed conference papers were eligible because much of the software-architecture literature is conference based. Conceptual descriptions without evaluation, purely operational studies after decomposition, and papers without relevant boundary or maintainability outcomes were excluded from the primary synthesis. Foundational books, secondary reviews, and Microsoft platform guidance were retained for theory or .NET interpretation but were not counted as primary empirical studies.

Fifty-eight unique candidate records were screened after de-duplication. Twenty-nine were excluded at title/abstract stage, leaving 29 reports for full-text assessment. Twelve were excluded because they lacked a relevant coupling/maintainability outcome, were conceptual-only, did not involve legacy/monolith decomposition, or duplicated a more complete report. Seventeen empirical studies entered the qualitative synthesis (Figure 1). Data extraction captured system context, evaluation design, decomposition signal, role of DDD/domain knowledge, coupling or cohesion outcome, maintainability/evolvability outcome, platform, and major limitations.

Methodological confidence was judged using criteria appropriate to software-engineering empirical work: clarity of the research question, transparency of the decomposition method, realism and size of the evaluated system, explicit comparator or baseline, appropriateness of outcome metrics, and discussion of validity or transferability. Studies were classified as high, moderate, or limited confidence rather than assigned a pseudo-precise numerical quality score. The purpose was to weight interpretation rather than exclude studies on the basis of method family.

Meta-analysis was not undertaken because coupling, cohesion, maintainability, and migration outcomes were operationalised differently across studies. Some studies reported structural metrics, others runtime modularity or execution-trace measures, and industrial cases relied on developer experience, development speed, or qualitative maintainability assessment. A concept-centric narrative synthesis was therefore used. Evidence was organised around semantic/domain alignment, static dependency control, runtime and data-flow validation, migration strategy, and downstream maintainability. Table 1 summarises the protocol.

3. Results

Seventeen primary studies published between 2016 and 2021 met the eligibility criteria (Table 2). The evidence comprised industrial migration reports, mixed-method practitioner studies, validated decomposition tools, static/dynamic analysis, execution-trace clustering, data-flow analysis, topic modelling, graph learning, and code-embedding approaches. Java systems dominated the automated-decomposition literature. No included peer-reviewed study directly evaluated DDD-led decomposition of a legacy .NET monolith. Two contemporaneous Microsoft sources were therefore used only to interpret implementation feasibility in .NET, not to increase the empirical study count [37,38]. Figure 3 shows the distribution of evidence themes.

DDD and business-domain knowledge were most useful at the initial boundary-discovery stage. Gysel et al. translated DDD concepts and coupling criteria into Service Cutter, demonstrating how entities, aggregates, and relationship weights can support service-cut decisions [22]. Krause et al. applied bounded-context reasoning to a real industrial legacy system but supplemented it with static and runtime visualisation because domain structure alone did not reveal all operational dependencies [23]. Stranner et al. used bounded contexts and facades in a large automotive ERP migration and reported improved developer preference and development speed as the migration progressed [31]. Kapferer and Zimmermann formalised strategic DDD through Context Mapper and reported validation through prototypes, action research, and case studies [33,34]. These findings converge on

DDD as a semantic structuring mechanism rather than a complete extraction algorithm.

Evidence on coupling and cohesion was stronger than evidence on long-term maintainability. Li et al. used business logic and data-flow models to identify candidates with sound coupling and cohesion [24]. Execution-trace studies showed that runtime behaviour can reveal functional relationships that source structure misses [25,26]. Mono2Micro combined business use cases and runtime calls to derive candidates and evaluated the approach on open-source and proprietary systems [27]. Graph and representation-learning approaches likewise optimised structural or semantic grouping [28-30]. Across methods, the recurring objective was the same: maximise internal cohesion while minimising dependency edges that cross the proposed service boundary.

The synthesis also showed why a low static edge count is insufficient. A service can be structurally separated yet remain coupled through runtime sequencing, shared state, or cross-service transactions. Krause et al. demonstrated that dynamic traces change the interpretation of boundaries derived from static information [23], while data-flow decomposition made datastore interactions part of the service-identification problem [24]. The practical implication is that DDD should define what belongs together semantically, while technical evidence tests whether those elements can change and execute together without extensive cross-boundary coordination.

Maintainability evidence was directionally positive but heterogeneous. Knoche and Hasselbring described smaller codebases and stronger separation as important mechanisms in legacy modernisation and reported practical experience from an industrial migration [18]. Fritzsch et al. found maintainability to be a major migration driver across 14 industrial systems, while also reporting difficulty finding suitable decomposition approaches [19]. The automotive bounded-context case reported markedly improved development speed and strong developer preference for the new services [31]. Da Silva et al. concluded from pilot and case work that modularisation may itself be a rational endpoint when the operational complexity of microservices is not justified [32]. These studies suggest that maintainability benefits arise from enforceable modular boundaries and change isolation rather than distribution alone.

Industry evidence further emphasised the socio-technical nature of the problem. Bogner et al. reported variability in microservice technologies and quality expectations across companies [20]. Waseem et al. found that practitioners commonly used DDD together with business-capability analysis and identified system complexity as a continuing design challenge [21]. Service ownership, organisational structure, release process, and observability therefore influence whether a technically coherent service actually reduces coordination cost.

The .NET-specific evidence through 2021 was primarily architectural guidance rather than comparative empirical research. Microsoft's 2021 .NET microservices guidance described bounded contexts as natural microservice candidates, advised keeping DDD-oriented services cohesive, and demonstrated domain models, aggregates, CQRS, and per-service data ownership in eShopOnContainers [37,38]. This guidance is consistent with the peer-reviewed findings but does not establish a .NET-specific causal effect. Accordingly, the transferable conclusion is architectural: C# projects, namespaces, Entity Framework DbContexts, stored procedures, and dependency-injection registrations should be treated as technical evidence to validate domain boundaries rather than as boundaries by default.

The relationship between boundary quality and coupling was clearer when the evidence was organised by dependency type. Structural coupling was represented by source references, calls, inheritance, imports, and shared components; behavioural coupling was represented by execution traces and business-use-case flows; data coupling appeared where responsibilities converged on the same datastore or transaction. The automated approaches differed in how they weighted these signals, but none demonstrated that a single measure was sufficient across systems [23-30]. Static-only clustering can overlook dynamic sequencing, whereas trace-only analysis may miss rarely executed but architecturally significant dependencies. Domain modelling supplies a further dimension by distinguishing an intentional business relationship from an accidental implementation dependency. A candidate service boundary is therefore strongest when semantic cohesion, structural separation, runtime behaviour, and data ownership point to the same grouping. Where those signals conflict, the conflict itself is a useful migration finding because it identifies a dependency that must be redesigned before distribution.

Change propagation provides the most direct bridge between coupling and maintainability. Classical coupling metrics explain how modules depend on one another [7,8], but a migration decision ultimately concerns whether future business changes can be implemented locally. Industrial migration reports associated microservices with smaller independently changeable units, while also describing decomposition difficulty and new distributed-system complexity [16-21]. This suggests two distinct failure modes. A boundary can be too coarse, leaving unrelated capabilities in one deployment and preserving broad regression scope; or it can be too fine, forcing ordinary feature work to coordinate changes across several services. The reviewed evidence does not provide a universal service size, but it consistently supports evaluating the frequency and stability of cross-boundary interactions. Stable read-only dependencies are less damaging to maintainability than bidirectional calls, shared writes, cyclic dependencies, or contracts that change whenever either side evolves.

Migration sequencing also affected the interpretation of outcomes. Several industrial reports described incremental extraction, facades, or staged architectural change rather than a single replacement of the monolith [17-19,31,32]. That sequencing matters methodologically because a system may obtain much of the maintainability benefit from better modularisation before any network boundary is introduced. If a bounded-context refactoring reduces change scope and clarifies ownership inside one deployable application, the incremental benefit of a separate service should be evaluated against its operational cost. Conversely, contexts that require independent scaling, deployment cadence, resilience, or team ownership provide a stronger reason for physical extraction. The evidence therefore supports distinguishing the quality of decomposition from the decision to distribute the decomposed parts. Figure 2 summarises this evidence-based mechanism connecting DDD boundary discovery with coupling and maintainability outcomes.

3.1 Methodological appraisal

The strongest evidence came from industrial cases that documented the migration context and from multi-system decomposition studies with explicit comparison metrics. Confidence was lower when a method was demonstrated on a single benchmark or when structural metrics were used as proxies for maintenance effort. No study was excluded for using a particular method family, but interpretation distinguished direct maintainability observations from inferred

maintainability based on coupling or cohesion. Table 3 summarises the principal strength and limitation of each evidence group.

4. Discussion

The principal conclusion is that DDD contributes to microservice decomposition by making business meaning explicit, not by eliminating the need for technical analysis. A bounded context is a hypothesis about semantic cohesion: concepts inside the boundary share a language, rules, and ownership model. The empirical literature shows that this hypothesis becomes more credible when static dependencies, runtime traces, data flows, and observed change patterns support the same separation [22-30]. For a legacy .NET application, the corresponding evidence sources include solution/project references, namespace dependencies, dependency-injection graphs, Entity Framework mappings, stored procedures, message contracts, telemetry traces, and repository co-change.

This distinction resolves a common decomposition error. Legacy code often mirrors historical layers rather than business capabilities. Splitting by current Visual Studio projects or technical layers can produce services that must collaborate on nearly every business use case. DDD can correct that by grouping responsibilities around domain capabilities, but the new boundary must still be tested for remote-call frequency and transaction scope. If a candidate context frequently requires synchronous calls to another context to satisfy a single user request, the two contexts may be too tightly coupled for independent deployment even if their domain names differ.

Data ownership is particularly important. A shared database can preserve strong coupling after code is split because schema changes, transactions, and write rules remain globally coordinated. Contemporaneous .NET guidance emphasised service-specific persistence in microservice examples [37,38], and data-flow decomposition research treated datastore relationships as part of boundary identification [24]. During migration, physical database separation need not occur immediately, but write ownership should become directional and explicit. One bounded context should control each invariant; other contexts should consume contracts, events, or purpose-specific read models rather than updating the same tables freely.

The evidence also supports a modular-monolith stage. Da Silva et al. observed that the operational complexity of microservices may not be justified after successful modularisation [32], and industrial migration work repeatedly described incremental transition rather than one-step replacement [17-19]. For legacy .NET, this suggests reorganising the existing application around bounded-context modules, introducing facades or interfaces, breaking dependency cycles, and measuring change locality before creating separate deployments. This stage reveals whether the desired maintainability benefit comes from better modularity alone or whether independent scaling, release cadence, fault isolation, or team ownership justify distribution. This incremental reasoning is also consistent with established architecture and refactoring principles, which treat boundaries and interfaces as design decisions that should evolve under controlled change rather than as a one-time physical split [41,42].

Maintainability should therefore be measured after decomposition rather than assumed. Useful indicators include the number of modules/services touched per feature, percentage of changes confined to one context, cross-context pull requests, build and test scope, contract-change frequency, defect localisation time, and developer-reported comprehension. Structural coupling metrics remain useful, but a low edge count can coexist with high coordination cost if the few remaining edges are semantically unstable or transactionally critical. Conversely, a bounded context with several stable read-only dependencies may still be maintainable if ownership and contracts are clear.

Service granularity is another boundary condition. Fine-grained services can improve internal cohesion but increase the number of network interactions, deployment units, and cross-team interfaces. Microservice reviews available within the evidence window documented architectural smells associated with inappropriate service boundaries and excessive coupling [13,36]. The practical objective is therefore not to maximise service count. A bounded context may remain one service, several coordinated services, or a module within a monolith depending on change patterns and operational needs.

For .NET specifically, the available evidence supports an architecture-level rather than tool-level conclusion. Microsoft's reference architecture showed that DDD, aggregates, CQRS, and per-service persistence were practical with ASP.NET Core and Entity Framework [37,38]. However, there was no mature peer-reviewed evidence demonstrating that a specific C# decomposition tool or DDD procedure reliably reduced coupling in legacy .NET applications. A rigorous .NET migration should therefore collect platform-specific dependency data and use DDD as the organising model rather than assuming results from Java-based clustering algorithms will transfer unchanged.

A future empirical study could address this gap by selecting several legacy .NET applications and comparing three decomposition conditions: DDD-only context mapping; static C# dependency clustering; and DDD combined with static dependencies, runtime traces, and database ownership. Before/after outcomes could include afferent/efferent coupling, change coupling from version history, number of cross-boundary database operations, build/test scope, feature lead time, and maintainability index. Such a design would directly test the claim implied by this review rather than extrapolating from heterogeneous platforms.

For an existing C# estate, boundary validation can be made concrete without requiring a specialised decomposition product. Compile-time evidence can be derived from solution and project references, namespace dependencies, public interfaces, shared NuGet packages, and dependency-injection registrations. Persistence evidence can be derived from Entity Framework DbContexts, repository implementations, stored procedures, table access, and transaction scopes. Runtime evidence can be gathered from representative HTTP requests, message handlers, scheduled jobs, and distributed traces. Version-control history adds a temporal view: files or projects that repeatedly change together may indicate a business dependency that is not obvious from current code structure. None of these signals should automatically dictate the service cut, but together they provide falsifiable tests of a proposed bounded context. A context that appears independent on a whiteboard but is crossed by most transactions or feature changes requires further redesign before extraction.

The organisational dimension should be evaluated with the same care. A microservice is intended to create an ownership boundary as well as a technical one. If each change to Service A routinely requires coordinated releases from the teams responsible for Services B and C, the system remains highly coupled even when source-code metrics look favourable. DDD's explicit vocabulary and context ownership can reduce this coordination burden by clarifying which team is authoritative for a model and its rules. Industrial studies within

the review support the connection between migration architecture, team practices, deployment processes, and perceived software quality [17,20,21,31]. Consequently, maintainability assessment should combine code-level measures with workflow evidence such as cross-team change requests, coordinated release frequency, ownership ambiguity, and the number of repositories touched by a feature. This socio-technical view prevents a superficially modular architecture from being judged successful while coordination cost simply moves from code to teams.

A practical migration sequence can be derived from these findings. First, domain experts and developers should establish a context map that identifies core, supporting, and generic responsibilities and documents the relationships among contexts. Second, the proposed boundaries should be challenged using source dependencies, runtime traces, database access, and historical co-change. Third, dependencies that violate the intended ownership model should be classified: some can be removed, some replaced by explicit contracts, and some temporarily isolated behind a facade or anti-corruption layer. Fourth, the boundaries should be enforced inside the existing deployment so that changes can be observed without simultaneously introducing network and operational complexity. Only after this period of validation should selected contexts be extracted as independently deployable services. This sequence is consistent with incremental migration reports and reduces the risk that a conceptual context map becomes a distributed monolith [17-19,31,32]. It also creates a measurable baseline: if modularisation has already improved change locality, the additional value of distribution can be judged against its costs rather than assumed in advance.

Interface design is the final coupling test. A service that exposes internal domain objects or requires callers to understand its persistence structure leaks implementation decisions across the boundary and weakens future evolvability. DDD instead favours contracts expressed in the language of the supplying context, with translation where neighbouring contexts use different models [4,5,33-35]. For .NET migrations, this implies keeping internal domain entities separate from public transport contracts, versioning integration messages deliberately, and avoiding shared domain-model assemblies across independently deployable services. The number of calls is therefore only one aspect of coupling; the stability and semantic independence of the contract are equally important. A coarse, stable interface can preserve autonomy even when two contexts communicate frequently, whereas a low-frequency interface that exposes internal schemas can create substantial maintenance risk whenever the supplier changes.

The review has limitations. First, the evidence base through 2021 was still relatively young and strongly weighted toward Java or language-independent studies. Second, maintainability was often inferred from coupling, cohesion, or developer experience rather than longitudinal maintenance cost. Third, automated methods used different metrics and datasets, precluding statistical synthesis. Fourth, DDD was applied at different depths: some studies used explicit bounded contexts while others relied on business capabilities or semantic information that is compatible with, but not identical to, DDD. These limitations reduce precision but not the consistent cross-method observation that semantic alignment is strongest when reinforced by technical dependency evidence.

Table 4 converts the synthesis into a practical evaluation framework suitable for a legacy .NET migration. It deliberately treats microservice extraction as conditional. The target is a maintainable boundary with clear domain and data ownership; independent deployment is justified only when it adds operational or organisational value.

5. Conclusion

The literature supports Domain-Driven Design as a strong semantic foundation for decomposing legacy applications, but not as a stand-alone algorithm for producing low-coupled microservices. Bounded contexts, subdomains, and ubiquitous language clarify business ownership; static dependencies, runtime traces, data flows, and migration constraints determine whether those boundaries are technically viable. Across industrial cases and decomposition methods, the most credible maintainability gains were associated with clearer ownership, change isolation, modularisation, and incremental migration rather than with service count alone. Accordingly, bounded contexts should be identified and validated with C# and runtime dependency evidence, explicit data ownership should be established, modularisation should precede distribution, and extraction should be limited to boundaries that demonstrate low change coupling and a clear need for independent deployment.

References

1. Page MJ, McKenzie JE, Bossuyt PM, Boutron I, Hoffmann TC, Mulrow CD, et al. The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. BMJ. 2021;372:n71. doi:10.1136/bmj.n71.
2. Kitchenham B, Brereton OP, Budgen D, Turner M, Bailey J, Linkman S. Systematic literature reviews in software engineering - a tertiary study. Inf Softw Technol. 2010;52(8):792-805. doi:10.1016/j.infsof.2010.03.006.
3. Petersen K, Feldt R, Mujtaba S, Mattsson M. Systematic mapping studies in software engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering. 2008. p.68-77.
4. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley; 2003.
5. Vernon V. Implementing Domain-Driven Design. Boston: Addison-Wesley; 2013.
6. Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM. 1972;15(12):1053-1058. doi:10.1145/361598.361623.
7. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Trans Softw Eng. 1994;20(6):476-493. doi:10.1109/32.295895.
8. Briand LC, Daly JW, Wüst J. A unified framework for coupling measurement in object-oriented systems. IEEE Trans Softw Eng. 1999;25(1):91-121. doi:10.1109/32.748920.
9. Newman S. Building Microservices. Sebastopol: O'Reilly Media; 2015.
10. Richardson C. Microservices Patterns. Shelter Island: Manning; 2018.
11. Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L. Microservices: yesterday, today, and tomorrow. In: Mazzara M, Meyer B, editors. Present and Ulterior Software Engineering. Cham: Springer; 2017. p.195-216.
12. Pahl C, Jamshidi P. Microservices: a systematic mapping study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science. 2016. p.137-146. doi:10.5220/0005785501370146.
13. Di Francesco P, Lago P, Malavolta I. Architecting with microservices: a systematic mapping study. J Syst Softw. 2019;150:77-97. doi:10.1016/j.jss.2019.01.001.
14. Soldani J, Tamburri DA, van den Heuvel WJ. The pains and gains of microservices: a systematic grey literature review. J Syst Softw. 2018;146:215-232. doi:10.1016/j.jss.2018.09.082.
15. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S. Microservices: the journey so far and challenges ahead. IEEE Softw. 2018;35(3):24-35. doi:10.1109/MS.2018.2141039.
16. Taibi D, Lenarduzzi V, Pahl C. Processes, motivations, and issues for migrating to microservices architectures: an empirical

investigation. IEEE Cloud Comput. 2017;4(5):22-32. doi:10.1109/MCC.2017.4250931.
17. Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw. 2016;33(3):42-52. doi:10.1109/MS.2016.64.
18. Knoche H, Hasselbring W. Using microservices for legacy software modernization. IEEE Softw. 2018;35(3):44-49. doi:10.1109/MS.2018.2141035.
19. Fritzsch J, Bogner J, Wagner S, Zimmermann A. Microservices migration in industry: intentions, strategies, and challenges. In: 2019 IEEE International Conference on Software Maintenance and Evolution. 2019. doi:10.1109/ICSME.2019.00081.
20. Bogner J, Fritzsch J, Wagner S, Zimmermann A. Microservices in industry: insights into technologies, characteristics, and software quality. In: 2019 IEEE International Conference on Software Architecture Companion. 2019. p.187-195. doi:10.1109/ICSA-C.2019.00041.
21. Waseem M, Liang P, Shahin M, Di Salle A, Márquez G. Design, monitoring, and testing of microservices systems: the practitioners' perspective. J Syst Softw. 2021;182:111061. doi:10.1016/j.jss.2021.111061.
22. Gysel M, Kölbener L, Giersche W, Zimmermann O. Service Cutter: a systematic approach to service decomposition. In: Service-Oriented and Cloud Computing. Cham: Springer; 2016. p.185-200. doi:10.1007/978-3-319-44482-6_12.
23. Krause A, Zirkelbach C, Hasselbring W, Lenga S, Kröger D. Microservice decomposition via static and dynamic analysis of the monolith. In: 2020 IEEE International Conference on Software Architecture Companion. 2020. p.9-16. doi:10.1109/ICSA-C50368.2020.00011.
24. Li S, Zhang H, Jia Z, Li Z, Zhang C, Li J, et al. A dataflow-driven approach to identifying microservices from monolithic applications. J Syst Softw. 2019;157:110380. doi:10.1016/j.jss.2019.07.008.
25. Jin W, Liu T, Zheng Q, Cui D, Cai Y. Functionality-oriented microservice extraction based on execution trace clustering. In: 2018 IEEE International Conference on Web Services. 2018. p.211-218. doi:10.1109/ICWS.2018.00034.
26. Jin W, Liu T, Cai Y, Kazman R, Mo R, Zheng Q. Service candidate identification from monolithic systems based on execution traces. IEEE Trans Softw Eng. 2021;47(5):987-1007. doi:10.1109/TSE.2019.2910531.
27. Kalia AK, Xiao J, Krishna R, Sinha S, Vukovic M, Banerjee D. Mono2Micro: a practical and effective tool for decomposing monolithic Java applications to microservices. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021. doi:10.1145/3468264.3473915.
28. Desai U, Bandyopadhyay S, Tamilselvam S. Graph neural network to dilute outliers for refactoring monolith application. Proc AAAI Conf Artif Intell. 2021;35(1):72-80. doi:10.1609/aaai.v35i1.16079.
29. Brito MA, Cunha J, Saraiva J. Identification of microservices from monolithic applications through topic modelling. In: Proceedings of the 36th ACM/SIGAPP Symposium on Applied Computing. 2021. p.1409-1418. doi:10.1145/3412841.3442016.
30. Al-Debagy O, Martinek P. A microservice decomposition method through using distributed representation of source code. Scalable Comput Pract Exp. 2021;22(1):39-52. doi:10.12694/scpe.v22i1.1836.
31. Stranner H, Strobl S, Bernhart M, Grechenig T. Microservice decomposition: a case study of a large industrial software migration in the automotive industry. In: Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering. 2020. p.498-505. doi:10.5220/0009564604980505.
32. da Silva HHOS, Carneiro GF, Monteiro MP. Towards a roadmap for the migration of legacy software systems to a microservice based architecture. In: Proceedings of the 9th International Conference on Cloud Computing and Services Science. 2019. p.37-47. doi:10.5220/0007618400370047.
33. Kapferer S, Zimmermann O. Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development. 2020. p.299-306. doi:10.5220/0008910502990306.
34. Kapferer S, Zimmermann O. Domain-driven service design: context modeling, model refactoring and contract generation. In: Service-Oriented Computing. Cham: Springer; 2020. p.189-208. doi:10.1007/978-3-030-64846-6_11.
35. Rademacher F, Sachweh S, Zündorf A. Towards a UML profile for domain-driven design of microservice architectures. In: Software Engineering and Formal Methods. Cham: Springer; 2018. p.230-245. doi:10.1007/978-3-319-74781-1_17.
36. Neri D, Soldani J, Zimmermann O, Brogi A. Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Softw-Intensive Cyber-Phys Syst. 2020;35:3-15. doi:10.1007/s00450-019-00407-8.
37. de la Torre C, Wagner B, Rousos M. .NET Microservices: Architecture for Containerized .NET Applications. Redmond: Microsoft Corporation; 2021.
38. Microsoft. Designing a DDD-oriented microservice. .NET Microservices Architecture guidance. 2021.
39. Lewis J, Fowler M. Microservices: a definition of this new architectural term. martinfowler.com; 2014.
40. Fielding RT. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine; 2000.
41. Kerievsky J. Refactoring to Patterns. Boston: Addison-Wesley; 2004.
42. Bass L, Clements P, Kazman R. Software Architecture in Practice. 3rd ed. Boston: Addison-Wesley; 2012.

Figures

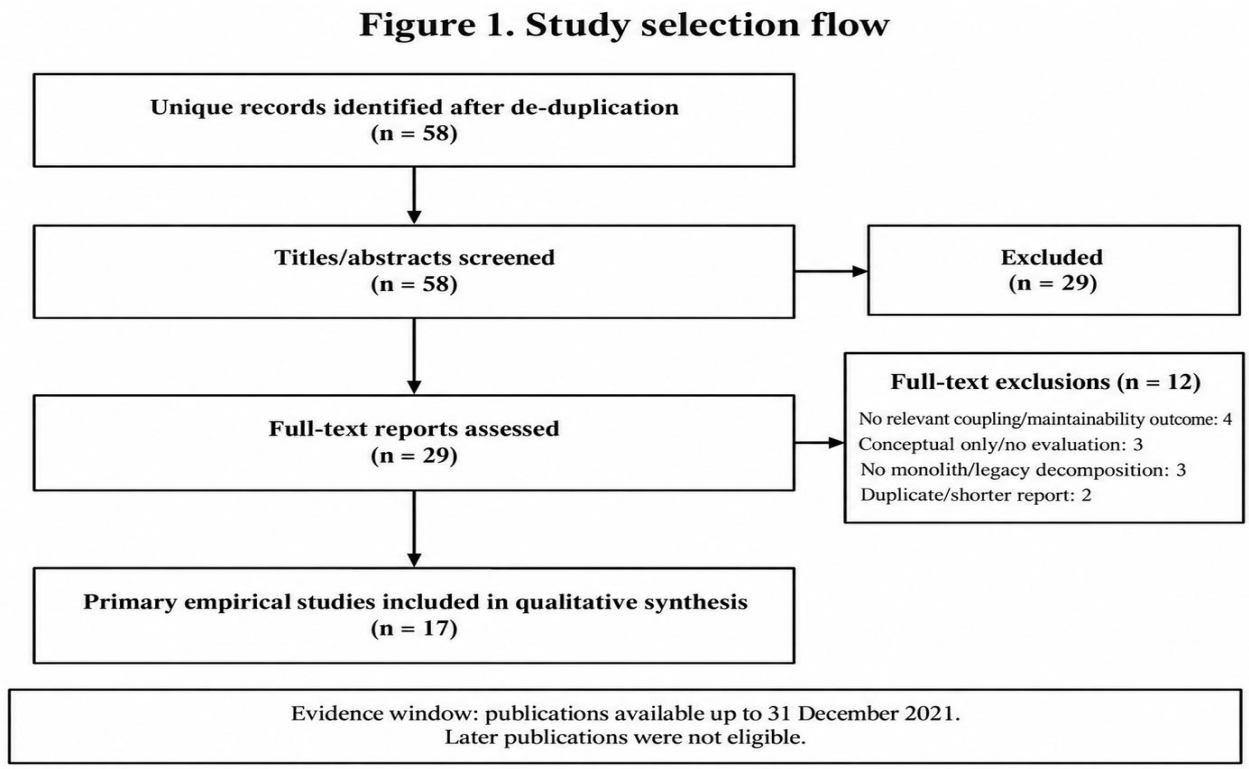

Figure 1. Study selection flow. The evidence window was limited to publications

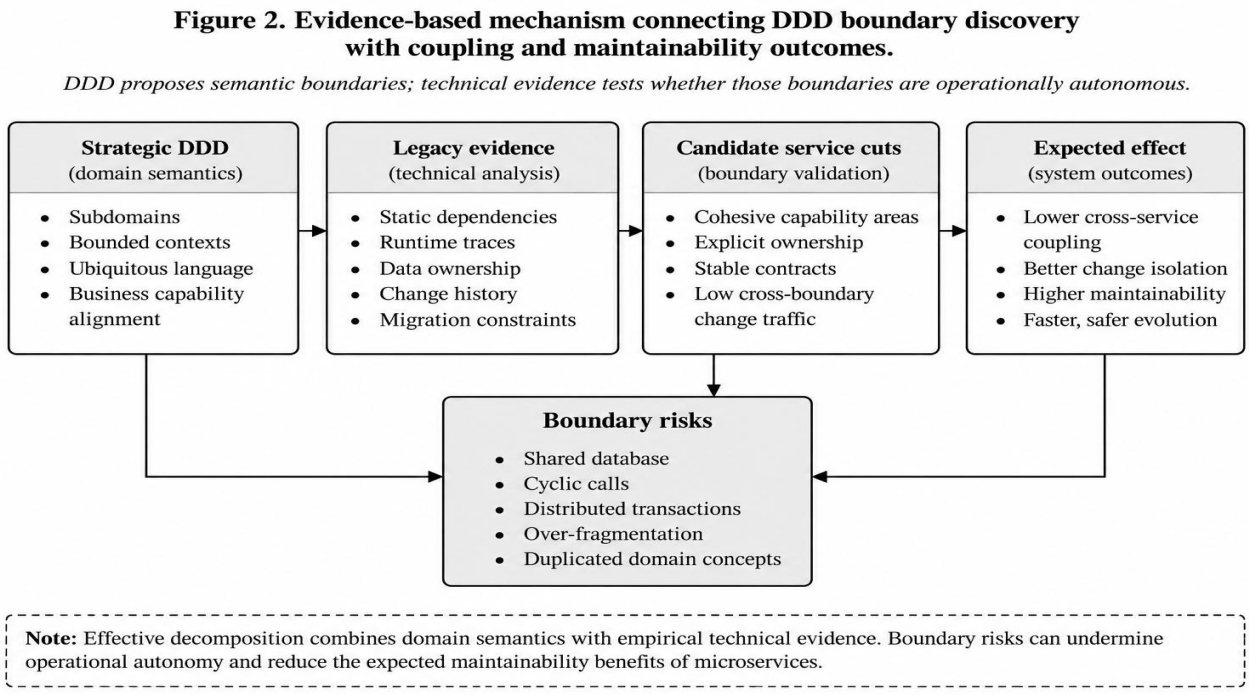

Figure 2. Evidence-based mechanism connecting DDD boundary discovery with coupling and maintainability outcomes.

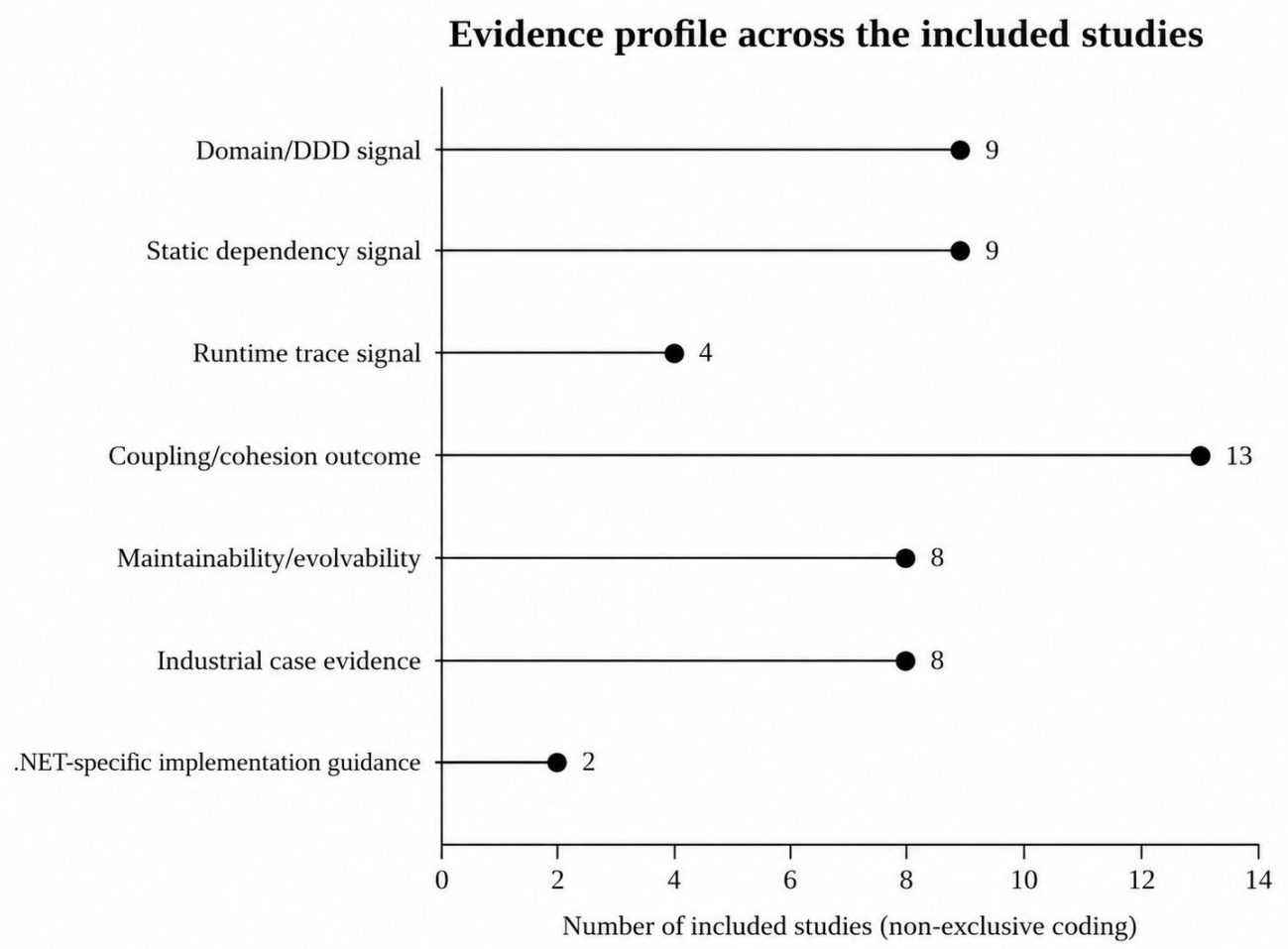

Figure 3. Evidence profile across the included empirical studies. Categories are non-exclusive.

Tables

Table 1. Eligibility criteria, search concepts, and synthesis plan.

Element	Specification
Evidence window	Database inception to 31 December 2021.
Population	Legacy or monolithic software systems; industrial migration studies; validated DDD/service-design approaches.
Intervention/exposure	DDD, bounded contexts, business capabilities, service identification/extraction, static/dynamic/data-flow decomposition.
Primary outcomes	Inter-service coupling, cohesion, modularity, maintainability, evolvability, change isolation, migration effort.
Sources	IEEE Xplore, ACM Digital Library, ScienceDirect, SpringerLink, Wiley and peer-reviewed conference repositories; citation chaining.
Primary inclusion	Empirical journal/conference study with a described method/system and interpretable boundary or maintenance outcome.
Primary exclusion	Conceptual-only papers; no legacy/decomposition context; no relevant outcome; duplicate/shorter version.
Synthesis	Structured qualitative synthesis; no pooled effect because systems, methods and metrics were heterogeneous.

Table 2. Characteristics and principal findings of included empirical studies

Study	Design/context	Boundary signal	Outcome focus	Principal finding
Balalaie et al., 2016 [17]	Industrial migration experience	Incremental microservice refactoring; business-aligned migration	Migration patterns; autonomy	Incremental decomposition and DevOps supported transition; architecture and organisation evolved together.
Gysel et al., 2016 [22]	Tool/method evaluation	DDD entities/aggregates + coupling criteria	Service cut; coupling	Service Cutter operationalised domain and coupling criteria for systematic decomposition.
Taibi et al., 2017 [16]	Industry investigation	Migration intentions and process	Maintainability; migration issues	Maintainability was a major driver; decomposition and distributed complexity were important concerns.
Knoche & Hasselbring, 2018 [18]	Industrial legacy modernisation	Incremental service decomposition	Maintainability; separation	Smaller codebases and strong separation were central modernisation mechanisms.
Jin et al., 2018 [25]	Multi-system trace clustering	Execution traces / functionality	Coupling; cohesion	Runtime behaviour identified functional relationships not reliably visible in source structure.
Li et al., 2019 [24]	Case-based method evaluation	Business logic + data-flow/process-datastore model	Coupling; cohesion	Data-flow-driven candidates achieved sound coupling/cohesion in the evaluated case.
Fritzsch et al., 2019 [19]	16 interviews; 14 systems	Industrial migration strategies	Maintainability; service cut	Maintainability/scalability drove migration; finding the right service cut was a major challenge.
Bogner et al., 2019 [20]	Industrial qualitative study	Microservice practice	Software quality; architecture	Reported diverse technologies and quality expectations; benefits depended on context.
da Silva et al., 2019 [32]	Pilot + case study	Legacy modularisation/migration roadmap	Maintainability; migration effort	Modularisation was a necessary stage and sometimes preferable to distribution.
Krause et al., 2020 [23]	Industrial legacy case	Bounded contexts + static/dynamic analysis	Boundary quality; dependencies	Runtime visualisation enriched domain/static analysis and exposed operational relationships.
Stranner et al., 2020 [31]	Large industrial ERP case	Bounded contexts + facades	Maintainability; development speed	Migration gained strong developer acceptance and improved development speed.
Kapferer & Zimmermann, 2020 [33,34]	Prototype/action research/cases	Strategic DDD + Context Mapper	Model clarity; service design	Machine-readable context maps supported iterative decomposition and service design.
Desai et al., 2021 [28]	Four monoliths / graph learning	Static dependency graph + GNN	Clustering quality	Graph learning improved decomposition metrics and highlighted

Study	Design/context	Boundary signal	Outcome focus	Principal finding
				refactoring outliers.
Jin et al., 2021 [26]	Six systems	Execution traces + functionality/modularity metrics	Modularity; evolvability	Service-candidate method improved several modularity/evolvability measures versus baselines.
Kalia et al., 2021 [27]	OSS + proprietary systems; survey	Business use cases + runtime calls	Cohesion; practical utility	Mono2Micro produced interpretable candidates and received positive practitioner feedback.
Brito et al., 2021 [29]	Topic-modelling evaluation	Source-code topics + structure	Service identification	Semantic vocabulary in code can recover functional groupings but depends on naming quality.
Al-Debagy & Martinek, 2021 [30]	Four applications	Code embeddings + hierarchical clustering	Domain/message cohesion	Embedding-based decomposition improved cohesion in most reported comparisons.

Table 3. Methodological appraisal of the included evidence.

Evidence group	Main strength	Main limitation	Interpretive confidence
Industrial migration studies [17-21,31,32]	High ecological validity; real organisational and legacy constraints.	Often observational; limited counterfactual comparison.	High-moderate
DDD/service-design tools [22,33,34]	Explicit decomposition criteria and traceable design rationale.	Tool validation smaller than enterprise longitudinal evaluation.	Moderate
Static/data-flow methods [24,28-30]	Reproducible metrics and automated comparison.	Structural/semantic proxies may not equal production maintainability.	Moderate-high
Runtime trace methods [23,25-27]	Capture operational behaviour invisible to source-only analysis.	Require representative workloads/traces; predominantly Java systems.	High for dependency discovery
.NET implementation guidance [34,35]	Direct platform relevance; concrete DDD and data-ownership examples.	Not peer-reviewed comparative evidence.	Contextual only

Table 4. Evidence-based decomposition framework for a legacy .NET application.

Decision	Evidence rationale	Recommended .NET assessment	Expected maintainability signal
Map bounded contexts first	DDD/industry studies support business-aligned ownership [21,22,31-35].	Workshops with domain experts; context map; ubiquitous language; capability ownership.	Fewer business features spanning unrelated modules.
Validate static dependencies	Automated studies show structural information changes candidate quality [24,28-30].	Project/assembly/namespace references; DI registrations; shared libraries; cycles.	Lower cross-context compile/change coupling.
Validate runtime behaviour	Trace studies expose relationships source analysis can miss [23,25-27].	Trace representative ASP.NET requests, jobs, messaging flows.	Shorter synchronous call chains; fewer cyclic runtime dependencies.
Assign data ownership	Data-flow evidence and .NET guidance emphasise autonomy [24,37,38].	Map DbContexts, tables, stored procedures and transaction scopes to contexts.	Most writes/invariants controlled by one context.
Modularise before extraction	Legacy migration cases support incremental change [17-19,32].	Enforce internal module APIs/facades while retaining one deployment initially.	Higher change locality before distributed-system cost is introduced.
Measure after changes	Maintainability benefits are not guaranteed by smaller services.	Track services touched per feature, contract churn, build/test scope, lead time.	Sustained reduction in coordination and regression scope.
Extract selectively	Microservices add operational complexity [13-15,36].	Extract contexts needing independent deployment, scaling, resilience or team ownership.	Distribution provides measurable value beyond modularity alone.