June 2026

Bridging Domain Models and Microservices: A Domain-Specific Language For Domain-Aligned System Design

Jasmin Fattah-Weil
University of Potsdam, jasmin.fattah-weil@wi.uni-potsdam.de

Marie Schmeißner
University of Potsdam, marie.schmeissner@uni-potsdam.de

Norbert Gronau
University of Potsdam, ngronau@lswi.de

BRIDGING DOMAIN MODELS AND MICROSERVICES: A DOMAIN-SPECIFIC LANGUAGE FOR DOMAIN-ALIGNED SYSTEM DESIGN

Completed Research Paper

Jasmin Fattah-Weil, University of Potsdam, Germany, jasmin.fattah-weil@wi.uni-potsdam.de

Marie Schmeißner, University of Potsdam, Germany, marie.schmeissner@uni-potsdam.de

Norbert Gronau, University of Potsdam, Germany, norbert.gronau@wi.uni-potsdam.de

Abstract

Microservice architectures offer flexibility but often widen the gap between business and IT understanding. While Domain-Driven Design (DDD) structures software around business domains, its implementation remains inconsistent and largely informal. This paper presents a domain-specific language (DSL) that formalizes DDD-based microservice architectures to improve strategic domain alignment and conceptual clarity. Following the Design Science Research Methodology, the study identifies recurring challenges in translating domain knowledge into executable microservice structures, derives modeling requirements from literature, and extends the Context Mapping Language (CML) accordingly. Based on Xtext, the DSL enables explicit representation of bounded contexts, context maps, and strategic DDD patterns, thus fostering shared understanding between business and technical stakeholders. The prototype is evaluated using a controlled reference scenario to demonstrate internal validity. The evaluation shows that DSL-based modeling enhances traceability and consistency in enterprise system design, contributing to process transparency and improved business–IT alignment.

Keywords: Microservices, Domain-Driven Design, Domain-Specific Languages, Software Architecture.

1 Introduction

The growing complexity of enterprise information systems forces organizations to revisit established architectural paradigms. Among the emerging responses, microservice architectures (MSA) have become a dominant design strategy, promising scalability, flexibility, and faster deployment cycles (Alshuqayran et al., 2016; Li et al., 2021; Newman, 2021). According to the Gartner Peer Community (2023), 74% of IT executives already use microservices in production, and the global cloud microservices market is projected to grow from USD 1.93 billion in 2024 to USD 11.36 billion by 2033 (Grand View Research, 2025). These trends illustrate the increasing strategic role of microservices in achieving future-proof IT architectures.

However, despite this momentum, organizations face persistent challenges in defining suitable service boundaries that reflect business semantics rather than purely technical decomposition (Taibi & Lenarduzzi, 2018). Domain-Driven Design (DDD) provides a conceptual foundation for addressing this challenge by structuring software around business domains (Evans, 2004). Yet, its practical implementation remains inconsistent. While DDD promotes bounded contexts and ubiquitous language, its artifacts are often informally documented and difficult to translate into executable microservice designs (Richardson, 2018; Waseem et al., 2021).

Since UML and similar notations cannot fully represent DDD's strategic and tactical patterns (Evans, 2004; Vernon, 2013), researchers have turned toward Domain-Specific Languages (DSLs) as lightweight, domain-oriented modeling instruments (Combemale et al., 2017; Rademacher et al., 2020;

Kapferer & Zimmermann, 2021). DSLs can reduce complexity and foster communication between business and IT stakeholders by enabling models that are both human-readable and technically precise (Fowler & Parsons, 2011; Combemale et al., 2017). In the context of microservices, DSLs promise to close the gap between conceptual DDD models and implementation-level architectures (Zhong et al., 2024; Sangabriel-Alarcón et al., 2023).

Nevertheless, current research offers no comprehensive analysis of how DSLs are applied to DDD-based microservice architectures, no systematic derivation of modeling requirements, and no prototype addressing the practical shortcomings in representing DDD artifacts, particularly bounded contexts and domain aggregates (Özkan et al., 2025; Brandolini, 2020; Millett, 2020).

To address this gap, this study investigates how DSLs can support the modeling of DDD-based microservice architectures. Following the Design Science Research Methodology (DSRM) (Peffers et al., 2007), the paper pursues three research questions:

- RQ1: Which approaches exist in the scientific literature for modelling DDD-based microservice architectures using DSLs?
- RQ2: Which modeling requirements must a DSL fulfill to support strategic and tactical DDD in microservice architectures?
- RQ3: How can an existing DSL be extended to address these requirements and improve the modeling of DDD-MSA artifacts?

This research aims to contribute a structured understanding of DSLs in the DDD-MSA context and to design a prototype that enhances conceptual consistency, reduces modeling ambiguity, and supports communication between domain experts and developers.

2 Background

2.1 Domain-Driven Design and Its Architectural Gap

DDD provides the conceptual foundation for aligning software with organizational knowledge. It introduces the domain model as a shared artifact between domain experts and developers and promotes collaboration through a ubiquitous language that bridges business and technical vocabularies (Evans 2004; Özkan et al. 2025). However, despite two decades of adoption, DDD remains theoretically rich but practically ambiguous. Its strategic patterns, bounded contexts, context maps, core domains, are rarely formalized, while teams focus on tactical elements such as entities and aggregates (Evans 2015; Vernon 2013; Millett 2020). The absence of precise modeling conventions leads to fragmentation, inconsistent terminology, and architectures drifting toward the “Big Ball of Mud” pattern (Foote & Yoder 1999; Brown et al. 1998). Without a mechanism to codify domain boundaries, organizations cannot ensure that software truly represents business semantics. Hence, DDD’s promise, to make business knowledge the organizing principle of software, remains only partially realized. This motivates research into formal, yet readable representations of strategic DDD concepts.

2.2 Microservices Expose the Limits of Informal DDD

Microservices have become the architectural response to growing system complexity, promoting autonomy, scalability, and agility (Lewis & Fowler 2014; Newman 2021). Their success, however, depends on correctly identifying service boundaries that correspond to business domains. In practice, these boundaries are often drawn from technical convenience rather than domain logic (Taibi & Lenarduzzi 2018; Waseem et al. 2021). DDD offers a theoretical guide for defining such boundaries, but because its models lack a formal syntax, microservice decomposition frequently relies on intuition and undocumented reasoning (Sangabriel-Alarcón et al. 2023). Consequently, even well-intended DDD-based microservices end up over-fragmented, redundant, or dependent on complex orchestration layers (Pautasso et al. 2017). The proliferation of distributed teams amplifies this risk: without shared, machine-interpretable models, communication gaps emerge between business analysts, architects, and

developers. Thus, the microservice paradigm intensifies the need for explicit, consistent, and semantically precise representations of DDD concepts.

2.3 DSLs Formalize DDD, but Integrative Approaches Are Missing

DSLs are lightweight, human-readable modeling languages tailored to narrow domains (Fowler & Parsons 2011; Combemale et al. 2017). They intentionally restrict scope to increase expressiveness and readability, enabling domain experts to participate in design without mastering general-purpose programming. In the DDD–MSA context, DSLs can capture aggregates, repositories, and Bounded Contexts in a formal syntax, bridging conceptual and technical levels (Rademacher et al. 2020; Kapferer & Zimmermann 2021). Recent work highlights the importance of domain-driven semantic modelling to integrate organizational and technical layers (Schmitz et al. 2025). Initial studies demonstrate that DSLs can enrich under-specified domain models and even generate code automatically (Sangabriel-Alarcón et al. 2023; Zhong et al. 2024; Mohottige et al. 2025). Yet none of the existing DSLs fully supports the breadth of strategic DDD patterns, particularly Distillation patterns and complex context relationships, nor do they offer an integrated solution spanning conceptual modelling, semantic consistency, and architectural reasoning. This gap motivates the present study. DDD articulates why domain alignment matters, MSA defines where this alignment must materialize, and DSLs represent a promising how, but the connection among them remains theoretically incomplete and empirically underdeveloped.

3 Methodology

This study follows a design science research approach (Peffers et al., 2007) aimed at developing and evaluating an artifact that addresses a key problem in enterprise software architecture: the lack of formal modeling instruments for DDD in the context of MSA. The research combines two complementary methods: (1) a Systematic Mapping Study (SMS) to analyze and structure the existing research landscape, and (2) a DSRM procedure to design, implement, and evaluate the proposed DSL.

3.1 Systematic Mapping Study

Following Petersen et al. (2008), the SMS explores the intersection of DDD, MSA, and DSLs to classify research and identify open issues. Compared with systematic literature reviews, mapping studies emphasize breadth and visualization over exhaustive synthesis—an appropriate choice for an emerging field marked by overlapping terminology. The study applies the four phases proposed by Petersen et al. (2008): planning, execution, analysis, and reporting.

3.1.1 Planning and Execution

guiding research question of the SMS (RQ1) is:

Which approaches exist in the scientific literature for modeling DDD-based microservice architectures using Domain-Specific Languages (DSLs)?

To ensure completeness, inclusion and exclusion criteria were defined. Included were English-language, peer-reviewed studies published between 2015 and 2025 that explicitly addressed DDD, MSA, and modeling aspects (IK1–IK4). Excluded were papers lacking modeling methodology, explicit DSL usage, or a DDD–MSA focus (EK1–EK4). The starting year 2015 was chosen because microservice architectures only gained significant academic attention after their emergence in the mid-2010s.

The search was conducted in four major academic databases: IEEE Xplore, ACM Digital Library, ScienceDirect, and SpringerLink ensuring coverage of both software engineering and information systems research. For each database, the search was applied to the title, abstract, and keyword fields. Two search strings were applied sequentially. The first used general modeling keywords to capture potentially relevant work, while the second explicitly included the term “Domain-Specific Language*”:

Search String 1: *("Microservice*" OR "Micro-service*" OR "MSA") AND ("Domain-driven Design" OR "DDD" OR "Domain driven design") AND ("model*" OR "design*" OR "visual*" OR "notation" OR "diagram*")*

Search String 2: *("Microservice*" OR "Micro-service*" OR "MSA") AND ("Domain-driven Design" OR "DDD") AND ("Domain-Specific Language*" OR "DSL")*

After retrieving the initial results, duplicate records were removed. Subsequently, a preliminary filtering step was conducted in which clearly irrelevant records were excluded prior to screening. The remaining sources were then screened based on titles and abstracts against the predefined inclusion and exclusion criteria. To complement the database search, backward and forward snowballing was conducted following Wohlin (2014). Forward snowballing was performed using Google Scholar to identify studies citing the included primary papers.

After the removal of duplicates and preliminary exclusions, 493 papers were screened in detail against the predefined criteria. Ultimately, 31 primary studies met all inclusion criteria and were retained for full-text analysis. The complete list of analyzed papers is available via the following link.

Figure 1 summarizes the identification and selection process, including the number of studies excluded at each stage.

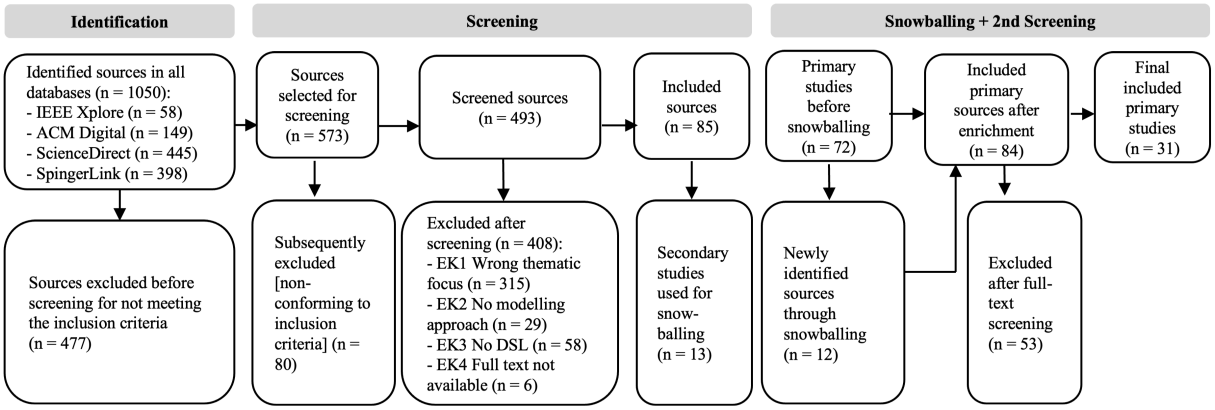

Figure 1 Identification and Selection Process of Primary Studies

3.1.2 Analysis Phase

The included studies were analyzed using the keywording approach proposed by Petersen et al. (2008). Keywords were extracted from titles, abstracts, and method sections to determine the main modeling purpose of each study. The resulting corpus was categorized along the DDD–DSL–MSA continuum, leading to the identification of four thematic clusters:

- Domain modeling and conceptualization (DDD-oriented)
- Formal DSL modeling (interface between DDD and MSA)
- Transformation into technical artifacts (MSA-oriented)
- Evaluation and feedback (bidirectional refinement)

Within these clusters, five DSL purposes were identified: model generation (MODEL-GEN), code generation (CODE-GEN), API generation (API-GEN), adaptation (ADAPT), and validation/check mechanisms (CHECK). These categories revealed the diversity of existing approaches and the absence of an integrative, domain-aligned modeling solution, thereby motivating the subsequent design phase.

3.2 Design Science Research Methodology

The second phase follows the DSRM framework by Peffers et al. (2007), which structures research activities around six iterative steps: (1) problem identification and motivation, (2) definition of objectives, (3) design and development, (4) demonstration, (5) evaluation, and (6) communication.

Problem Identification and Objective The SMS revealed a lack of formalized DSLs that can represent DDD constructs and bridge the gap between conceptual domain models and technical MSA implementations. Based on this insight, the research defines three guiding questions (RQ1–RQ3)

addressing the identification of existing approaches, derivation of modeling requirements, and prototype extension of a suitable DSL to close the identified gaps.

Design and Development In the design phase, the DSL type identified as most promising through the systematic mapping study -Context Mapping Language (CML) - was selected for extension. This choice was based on its comparatively broad coverage of strategic DDD patterns and its established use in both academic and industrial contexts. The derived requirements from RQ2 guided the design of targeted language extensions, focusing on closing the identified gaps in modeling strategic DDD constructs such as Separate Ways, Big Ball of Mud, and the Distillation Patterns (Highlighted Core, Segregated Core, Abstract Core, Cohesive Mechanisms). The resulting DSL metamodel formalizes key DDD constructs such as Context Map, Core Domain and Bounded Contexts. The implementation uses Xtext, a framework that supports textual syntax definition, parsing, and validation. The developed DSL enables consistent, domain-aligned modeling of domain structures and relationships within microservice architectures.

Demonstration and Evaluation The DSL was demonstrated in an illustrative case scenario representing a typical domain within enterprise software development. The evaluation assessed the expressiveness, structural consistency, and alignment of the DSL with DDD principles. Qualitative feedback focused on whether the language reduces modeling ambiguity and supports clearer communication between business and technical stakeholders.

4 Results of the Systematic Mapping Study

The systematic mapping study yielded 31 primary studies that explicitly employ DSLs in the context of DDD and MSA. These studies were categorized according to their primary modeling purpose, resulting in five functional clusters: Model Generation (MODEL-GEN), Code Generation (CODE-GEN), API Generation (API-GEN), Adaptation (ADAPT), and Validation (CHECK). To provide conceptual grounding for these clusters, each requirement (R100–R631) was mapped to one or more DSL capabilities based on the keywording and coding process described in Section 3. Table 1 presents the resulting categories, their definitions, exemplary requirements, and coding rules.

4.1 Distribution and Temporal Trends

Category	Definition	Example Requirement
MODEL-GEN	This category includes DSL functionalities for the explicit representation of business and technical domain terms and concepts as modelable constructs. The goal is the structural depiction and documentation of relevant artefacts at the model level.	R301: “The DSL shall model the strategic DDD pattern Bounded Context” – the DDD pattern Bounded Context must be explicitly representable in the DSL.
CODE-GEN	This category refers to the DSL’s capability to automatically transform modeled DDD concepts into technical artefacts, such as source code or API structures.	R601: “The DSL shall transform modeled Entities into context-specific MSA artefacts” – a DDD pattern must be transformed into a technical MSA artefact.
CHECK	This category includes explicit DSL mechanisms for analysis, validation, and model-based decision support, e.g., for identifying Bounded Contexts or evaluating pattern conformance.	R401: “The DSL shall provide visual or structural aids for recognizing context boundaries” – the DSL must include a function to support the identification of Bounded Context boundaries.

Table 1. Mapping of DSL to Modeling Capabilities.

Keywording and Cluster Formation

Following the keywording procedure (Petersen et al., 2008), the 31 included studies were categorized into five clusters according to their dominant purpose within the DDD–MSA context.

Each cluster represents a distinct type of DSL functionality, characterized by typical keywords (see Table 2).

Cluster	Purpose	Top Keywords
ADAPT	Migration of monolithic systems or architectural adaptation of existing processes, technologies, or systems toward concrete DDD–MSA realization.	integration, migration, refactoring
CHECK	Formal or automated validation/evaluation of architectural decisions, interface conformance, and model–code consistency in the DDD–MSA context.	conformance, assessment, patterns
MODEL-GEN	Creation of visual or textual (semi-)formal DDD or intermediate models for documentation, visualization, or reflection purposes (non-executable).	Context Map, metamodel, UML
CODE-GEN	Generation of executable MSA source code or technical microservice artefacts based on DSL models.	code generation, tool, automatic
API-GEN	Generation of structured API definitions from DSL models, typically transforming DDD concepts into service interfaces.	API, endpoint, Jolie

Table 2. Identified DSL Clusters and Assigned Studies.

Distribution and Temporal Trends

As shown in Figure 2, most publications fall into the MODEL-GEN category (29 %), followed by CHECK (23 %), ADAPT (19 %), and CODE-GEN (19 %). API-GEN accounts for only 10 %, indicating limited research on DSL-supported interface derivation.

The dominance of MODEL-GEN and CHECK reflects a strong academic focus on conceptual modeling and validation mechanisms rather than automation or deployment. A temporal analysis (Figure 3) shows that DSL research in the DDD–MSA context evolved in parallel with the broader adoption of microservices. Early studies (around 2020) primarily addressed visual and semi-formal modeling approaches (MODEL-GEN), while later works expanded toward executable artefacts, with CODE-GEN and API-GEN gaining visibility around 2022.

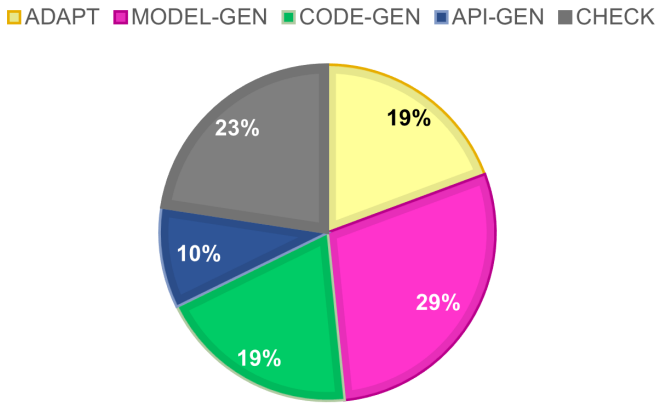

Figure 2. Distribution of publications across DSL categories.

The most recent trend (2023–2025) is the emergence of CHECK-oriented approaches emphasizing model–code consistency and architectural conformance. The ADAPT category spans the entire timeline (2018–2025), reflecting its role in migration and refactoring scenarios.

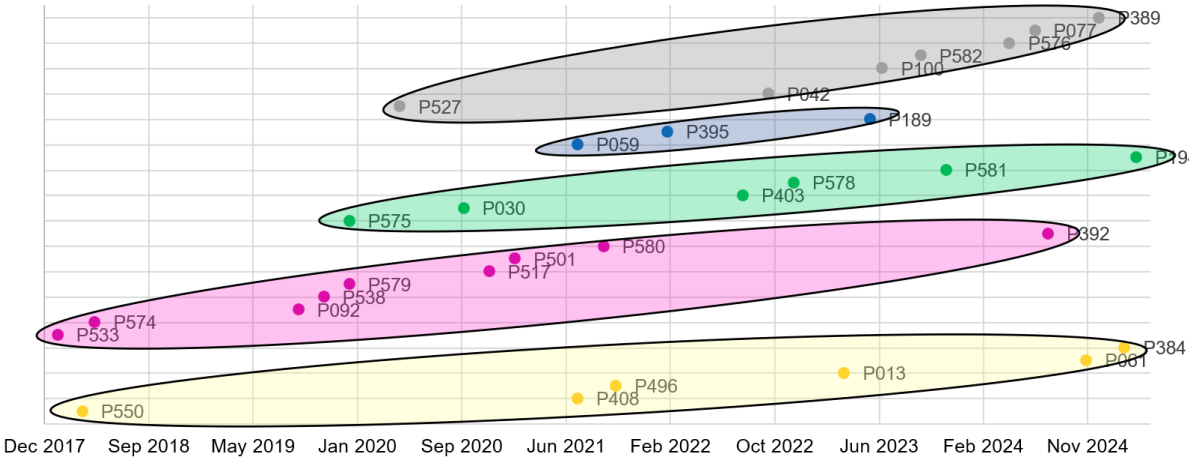

Figure 3. Publication timeline of the included references (Colours based on cluster from Fig. 2).

4.2 Thematic Insights

The five DSL categories collectively depict a research continuum from conceptual abstraction to technical realization:

- MODEL-GEN studies focus on generating semi-formal domain or context models to enhance conceptual clarity and support communication between business and technical stakeholders.
- CODE-GEN and API-GEN approaches operationalize DDD constructs, transforming domain models into technical artifacts or service interfaces.
- CHECK DSLs emphasize the automated validation of model integrity, ensuring coherence between domain models and implemented services.
- ADAPT DSLs assist in migration and refactoring tasks, bridging legacy architectures and microservice implementations.

The temporal and functional clustering suggests a maturation trajectory within the field: from modeling and visualization toward automation and quality assurance. However, while the categories were designed to capture distinct DSL foci, few approaches combine domain semantics with executable microservice generation. This observation points to a persistent fragmentation between conceptual and technical perspectives in current DSL research.

5 Artifact Design and Demonstration

5.1 Design Science Process

To answer RQ3, the study applies the DSRM (Peffers et al., 2007). Based on the mapping study and the derived requirements, the Context Mapping Language (CML) was selected as the design baseline. CML was chosen because it provides the broadest coverage of strategic DDD constructs among the evaluated DSLs, including Bounded Contexts, Context Maps, and key relationship patterns. In contrast, LEMMA focuses primarily on technical aspects and lacks support for strategic and distillation patterns, while SoaML and UML profiles offer only limited expressiveness for DDD semantics. Furthermore, CML spans multiple SMS-purpose categories (MODEL-GEN, CHECK, parts of CODE-GEN) and is designed as an extensible, lightweight core language, making it suitable for targeted enhancements without breaking compatibility. Its close alignment with DDD terminology and existing partial support for patterns such as Separate Ways and Distillation patterns further reinforce its suitability. Accordingly, CML serves as the most appropriate foundation for extending DSL-based support for strategic DDD modeling.

5.2 Problem Identification and Objectives

The mapping study revealed that while CML is widely used for modeling strategic DDD concepts, it lacks constructs for several strategic patterns defined by Evans (2015). These omissions limit its expressiveness in representing domain relationships and architectural decisions. Hence, the objective was to extend CML to include six previously unsupported patterns: Separate Ways, Big Ball of Mud (BBoM), Highlighted Core, Segregated Core, Abstract Core, and Cohesive Mechanisms. The design focuses on the MODEL-GEN level, i.e., conceptual modelling, rather than code generation or validation. This ensures a solid conceptual foundation for later extensions toward automation and conformance checking. Each new pattern addresses distinct requirements of domain modeling, ranging from documenting intentional separations (Separate Ways) to capturing shared abstractions across contexts (Abstract Core).

5.3 Artifact Design

Metamodel Extension: The extension began with a modification of the existing CML metamodel implemented in Xtext. CML was selected because it offers the strongest support for tactical and strategic DDD modeling compared to LEMMA and SoaML, while remaining sufficiently lightweight to allow extensibility. The extension targets the conceptual layer (MODEL-GEN), which is a prerequisite for future CHECK and CODE-GEN capabilities. To integrate the six strategic DDD patterns, new entities and relationships were added while preserving full backward compatibility with existing CML constructs. Separate Ways was introduced as a symmetric relationship type, complementing existing relationships such as Partnership and Shared Kernel. Big Ball of Mud (BBoM) and Abstract Core were positioned at the Context Map level to express cross-context dependencies and structural erosion patterns, whereas Highlighted Core, Segregated Core, and Cohesive Mechanisms were embedded within individual Bounded Contexts to represent internal strategic distinctions. In addition to structural extensions, the metamodel incorporates explicit semantic constraints. These constraints ensure, for example, that (a) Separate Ways cannot coexist with Partnership or Shared Kernel between the same contexts, (b) Highlighted Core must be unique within a context, (c) Segregated Core requires a corresponding non-core subdomain, and (d) a Big Ball of Mud cannot introduce cyclic strategic relationships. These validation rules were implemented through modular Xtext validators, enabling machine-checkable consistency without affecting existing syntax or transformation logic. The final metamodel integrates all six strategic DDD patterns while maintaining structural alignment with core DDD principles, extensibility of the original CML language, and forward compatibility with future generation and validation mechanisms.

Syntax Integration: New grammar rules were implemented in ContextMappingDSL.xtext. Each pattern received dedicated syntax elements, enabling explicit representation of its semantics. For instance, *CustomerManagement [SW] <-> [SW] ClaimsManagement* expresses a Separate Ways relationship, while BigBallOfMud LegacyCRM_BBoM contexts (LegacyCRM) marks a legacy subsystem. Optional parameters such as reason, criteria, or scope allow modelers to document decision rationales, thus increasing transparency. All extensions conform to the Apache License 2.0 requirements of the open-source ContextMapper project.

Semantic Constraints: To maintain model consistency, the extended DSL includes new validation rules implemented in Java, as shown in Table 3. Examples include:

- *Abstract Core* must reference at least two Bounded Contexts that are not linked by Separate Ways.
- *Separate Ways* relationships are exclusive, no parallel Partnership may exist between the same contexts.
- *Highlighted Core* elements must already exist within the referenced Bounded Context, while *Segregated Core* requires at least one core or supporting element.

These constraints ensure internal consistency and adherence to DDD semantics.

Pattern	Constraint Description	Affected Java Class(es)
Separate Ways	Exclusivity between two Bounded Contexts. No additional relationship pattern may exist between the same pair of Bounded Contexts.	BoundedContextRelationshipSemanticsValidator.java, ValidationMessages.java
Big Ball of Mud	The pattern must include at least one Bounded Context ("affectedContext").	ContextMapSemanticsValidator.java, ValidationMessages.java
Big Ball of Mud	All referenced relationships ("affectedRelationships") must involve the Bounded Contexts listed under "affectedContext."	ContextMapSemanticsValidator.java, ValidationMessages.java
Abstract Core	At least two participating Bounded Contexts ("participants") must be specified.	ContextMapSemanticsValidator.java, ValidationMessages.java
Abstract Core	The "participants" may not simultaneously be connected through the Separate Ways pattern.	ContextMapSemanticsValidator.java, ValidationMessages.java
Highlighted Core	All listed elements ("highlightedElements") must exist within the affected Bounded Context.	BoundedContextSemanticsValidator.java, ValidationMessages.java
Highlighted Core	At least one element ("highlightedElements") must be defined.	BoundedContextSemanticsValidator.java, ValidationMessages.java
Segregated Core	At least one element must be defined on either side ("coreElements" or "supportingElements").	BoundedContextSemanticsValidator.java, ValidationMessages.java
Segregated Core	All referenced elements must exist within the same Bounded Context.	BoundedContextSemanticsValidator.java, ValidationMessages.java
Segregated Core	The elements listed under "coreElements" and "supportingElements" must not overlap.	BoundedContextSemanticsValidator.java, ValidationMessages.java
Cohesive Mechanisms	If the scope is set to "MULTI_BC," multiple Bounded Contexts must be involved.	BoundedContextSemanticsValidator.java, ValidationMessages.java

Table 2. New CML Constraints for the Extended Strategic DDD Patterns.

5.4 Demonstration

The prototype was demonstrated and evaluated using the well-established "Lakeside Mutual" insurance case from the Context Mapper repository. The extended DSL was used to model several strategic decisions that are typically left implicit in DDD–MSA projects. Separate Ways was applied to document the intentional decoupling between CustomerManagement and ClaimsManagement, making explicit a dependency that had previously only been assumed. Big Ball of Mud (BBoM) captured the integration of a legacy CRM system, formalising its treatment as a read-only subsystem accessed through an ACL, an archetypical migration setting in enterprise environments. Within the Customer Core domain, the patterns Highlighted Core and Segregated Core were used to differentiate core business logic from supporting components, thereby sharpening the rationale for domain boundaries. Cohesive Mechanisms enabled the modelling of reusable computational logic such as the Risk Pricing Engine, while Abstract Core represented shared abstractions (e.g., CustomerId, PolicyId) across several Bounded Contexts to

improve semantic consistency. Together, these applications illustrate how the extended CML makes strategic DDD decisions explicit, traceable, and structurally verifiable.

5.5 Evaluation and Reflection

The evaluation follows a demonstration-oriented approach typical for early-stage Design Science Research (Peffers et al., 2007; Hevner et al., 2004). The proposed DSL extension was instantiated and applied to the "Lakeside Mutual" reference model from the Context Mapper repository in order to assess its ability to represent strategic Domain-Driven Design (DDD) patterns and enforce structural constraints.

All six new patterns were successfully instantiated and validated through the implemented constraints. Based on the Lakeside Mutual model, the evaluation confirms that the extended CML improves expressiveness and traceability in documenting architectural decisions. Separate Ways enabled explicit modeling of intentional decoupling between Bounded Contexts, while Highlighted Core and Segregated Core enhanced domain transparency by clarifying distinctions between core and supporting components. Their partial overlap proved complementary, offering different perspectives for domain experts and architects.

Big Ball of Mud and Abstract Core (Vernon, 2013; Evans, 2015; Millett & Tune, 2022) supported the identification of legacy entanglements and structural inconsistencies, whereas Cohesive Mechanisms captured reusable computational logic linking domain and technical layers. Collectively, these extensions enhanced the granularity and interpretability of domain-driven models—key goals in Domain-Driven Design and Model-Driven Engineering (Rademacher et al., 2020; Kapferer & Zimmermann, 2021).

A comparative overview of DSL requirement fulfilment (R100–R631) across SoaML, UML Profile, LEMMA, and CML is available in the supplementary material (available via the following link). This mapping confirms that CML provides the broadest coverage of strategic DDD constructs such as Bounded Contexts, Context Maps, and relationship patterns (Partnership, Anticorruption Layer), yet still lacks full support for Distillation Patterns and complex inter-context relations (Sangabriel-Alarcón et al., 2023; Zhong et al., 2024). Overall, the extended CML helps close the methodological gap between conceptual and technical design by embedding DDD semantics in a machine-verifiable structure.

The evaluation presented here focuses on a controlled demonstration scenario and therefore primarily assesses conceptual correctness and modeling expressiveness rather than practical usability. While this approach is common in early design science artifacts, further empirical validation is required to evaluate how practitioners adopt and interpret the proposed DSL constructs in real-world architecture design processes.

Future research should therefore examine the usability and scalability of the DSL in industrial settings, for example through practitioner workshops, architectural design studies, or comparative modeling experiments. Empirical validation with practitioners will be required to assess whether the DSL effectively improves communication between business and IT stakeholders. In addition, the DSL could be extended toward visual modeling representations and automated artifact generation (Mohottige et al., 2025).

Future research may also investigate how the DSL could represent data ownership and consistency boundaries across microservices, for instance to capture shared database dependencies or data consistency constraints between Bounded Contexts. Integrating collaborative modeling techniques such as Event Storming may further enhance domain understanding during early design phases while maintaining the formal rigor of DSL-based modeling.

Nevertheless, internal validity is high, as all implemented patterns and constraints are grounded in established DDD theory (Evans, 2015; Vernon, 2013). Furthermore, compared to existing DSL ecosystems for microservices modeling, the proposed extension explicitly focuses on representing strategic DDD patterns that are typically not formalized in current modeling tools.

6 Discussion

This study set out to explore how DSLs can support the modeling of DDD-based MSA and, more specifically, how existing DSLs and an extended Context Mapping Language (CML) can address documented practice gaps at the DDD–MSA boundary. Building on the SMS and DSRM phases, the findings are discussed in relation to the three research questions.

6.1 Interpreting the DSL landscape for DDD–MSA (RQ1)

RQ1 asks: *Which approaches exist in the scientific literature for modeling DDD–based microservice architectures using Domain-Specific Languages (DSLs)?*

The mapping study identifies three overarching types of DSL approaches used in the DDD–MSA context:

DDD-oriented modelling: These DSLs focus on expressing strategic and tactical DDD concepts such as Bounded Contexts, Context Maps, Aggregates, and Subdomains. Examples include Context Mapper (CML) and several metamodel-based approaches that formalize DDD structures for documentation, analysis, or communication.

Transformation-oriented: DSLs for deriving microservice artefacts These approaches use DSLs to transform DDD models into technical MSAs, such as service contracts, deployment descriptors, or code templates. Examples include model-to-code pipelines, DevOps-enabled modeling languages, and DSL extensions for API generation (e.g., Jolie-based approaches).

Analysis- and validation-oriented: DSLs These DSLs provide mechanisms for consistency checking, detection of architectural smells, evaluation of service boundaries, or conformance between domain models and implemented microservices. This includes DSLs aimed at assessing dependency structures, interface correctness, or propagation effects in service landscapes.

Across these types, the 31 analyzed studies map to five functional purposes - **MODEL-GEN, CODE-GEN, API-GEN, ADAPT,** and **CHECK** - which together demonstrate that DSLs are used either to represent DDD concepts, transform them into microservice artefacts, or validate architectural decisions. Core DSLs such as CML and LEMMA span multiple purposes, whereas narrower languages such as AjiL or TASL address specific subtasks (e.g., migration or API definition). SoaML- and UML-profile extensions appear in several studies but only partially support strategic DDD concepts. Thus, the answer to RQ1 is that existing approaches fall into three major categories: DDD modelling, transformation, and validation. Each category contributes to specific stages of the DDD–MSA modeling pipeline, yet no existing DSL provides an integrated, end-to-end solution.

6.2 Requirements for DSLs and the relative suitability of existing languages (RQ2)

By synthesising DDD, MSA, and DSL literature with repeatedly reported practice issues, the study derives six main requirements and 65 sub-requirements that a DSL must fulfil to support DDD-based microservice architectures. These requirements fall into three categories:

- **Capture domain and strategic design explicitly (MODEL-GEN)** – explicit modelling of strategic DDD concepts such as Bounded Contexts, Context Maps, Core Domain, and Distillation patterns.
- **Bridge from domain models to microservice artefacts (CODE-GEN)** – the ability to describe technical artefacts (e.g., API contracts, persistence strategies) in a form suitable for automated transformations.
- **Support systematic assessment and consistency checking (CHECK)** – mechanisms for consistency checking, conformance validation, and detection of architectural problems (e.g., MSA smells).

These requirements consolidate prior observations that DDD's strategic patterns are rarely formalised, that transformations from domain models to MSAs are insufficiently supported, and that modelling approaches lack explicit semantic validation (Schmidt & Thiry 2020; Sangabriel-Alarcón et al. 2023; Zhong et al. 2024). The resulting catalogue (R100–R631) is provided via the following OSF repository.

Evaluating existing DSLs against these requirements shows a clear result: CML is the only DSL that provides substantial coverage across all three requirement categories. It models most strategic DDD constructs, supports early validation through semantic checks, and integrates with a toolchain that enables partial code and diagram generation. In contrast, alternative DSLs (e.g., LEMMA, UML profiles, specialised DSLs) either lack strategic modelling capabilities or focus narrowly on technical artefact generation, leaving key DDD semantics unrepresented. Thus, CML emerges as the most suitable foundation for extending DSL support for DDD-MSA modelling, as it already fulfils a large subset of the identified requirements and can be systematically extended to close the remaining gaps, particularly in Distillation patterns and complex inter-context relationships.

6.3 Extending CML with strategic DDD patterns (RQ3)

RQ3 addressed the design question: How can the DSL identified in RQ2 (CML) be prototypically extended to better address practice gaps in the application of DDD in MSAs?

The DSRM intervention focuses deliberately on the **MODEL-GEN** side, following the observation that robust code generation and checking both presuppose a sufficiently expressive conceptual model. Concretely, CML is extended with six previously omitted strategic DDD patterns: **Separate Ways, Big Ball of Mud, Highlighted Core, Segregated Core, Abstract Core, and Cohesive Mechanisms** as can be seen in figure 3 in the supplementary material (OSF link), which illustrates the extended metamodel integrating these patterns within existing DDD concepts (e.g., Context Map, Bounded Context, and Relationship types). The complete metamodel, including all entities, relationships, and Xtext grammar definitions, is available via the OSF repository.

These are integrated at the metamodel level, reflected in the Xtext grammar, and guarded by new semantic constraints in the validation logic.

The extended CML is then applied to the *Lakeside Mutual example*. The demonstration and constraint checks show that all six patterns can be modeled consistently, and more importantly that their explicit representation changes how the architecture is reasoned about. For example:

- Separate Ways allows the language to distinguish between genuinely independent Bounded Contexts and relationships that are merely "missing" because they were never discussed. In the case study, making Separate Ways explicit led to the discovery of a previously overlooked potential dependency between *CustomerManagement* and *CustomerSelfService*.
- BBoM, anchored at context-map level, provides a formal way to label problematic legacy areas such as a monolithic CRM system and to connect them to ACL-based integration strategies. This reflects typical real-world migration scenarios that DDD and MSA literature identify as high-risk sources of erosion and coupling.
- Highlighted Core and Segregated Core help distinguish between core domain elements and supporting infrastructure within a Bounded Context, making the rationale for boundaries and prioritisation transparent, an issue repeatedly emphasised in DDD practice essays.
- Cohesive Mechanisms and Abstract Core serve as bridges between domain and technical views: they capture reusable technical mechanisms and cross-context abstractions (e.g., a consistent notion of customer identity) in a way that supports both Ubiquitous Language and potential future automation.

Conceptually, these extensions respond directly to critiques by Evans, Brandolini, Millett, and others that strategic DDD is often treated informally and left undocumented, which in turn complicates communication and evolution. By *lifting* these patterns into a formally defined DSL, with syntax,

semantics, and constraints, the extended CML provides a more faithful representation of how experienced DDD practitioners already think and talk about complex domains.

In sum, RQ3 is answered by demonstrating that CML can be feasibly and coherently extended to model central strategic DDD patterns, thereby increasing the expressiveness and transparency of DDD–MSA models without changing CML's core philosophy.

6.4 Overall Implications

In summary, the three RQs position DSLs as a **central integration mechanism** at the intersection of DDD and MSAs. The mapping study shows that DSLs already underpin many state-of-the-art approaches, but also that current languages distribute their strengths unevenly across conceptual, generative, and evaluative concerns. The requirements analysis provides a structured lens to reason about these trade-offs and to design next-generation DSLs that better support strategic DDD and end-to-end model-driven development. The CML extension illustrates how such evolution can be realised in practice and how encoding strategic patterns in a DSL can surface tacit assumptions, reveal blind spots, and document architectural decisions more systematically.

For research, this suggests several directions: extending CHECK and CODE-GEN capabilities atop richer conceptual models; integrating visual tooling to make strategic patterns accessible beyond DDD experts; and empirically studying how extended DSLs influence design quality, team communication, and the occurrence of MSA smells in real projects.

For practice, the work underlines that the choice and configuration of a DSL is not a purely technical tooling decision. It encodes which dimensions of DDD and MSA are visible, discussable, and automatable, and which remain implicit. By strengthening the strategic dimension in CML, the study provides a concrete step towards DSLs that not only generate microservice code, but also help architects and teams reason more critically about why their services are cut the way they are, and how those decisions relate back to domain models, legacy constraints, and long-term evolvability.

6.5 Future Research

The presented artifact represents an initial step toward more formalized modeling of Domain-Driven Design in microservice architectures. Future research should therefore focus on evaluating the practical applicability of the proposed DSL extension in real-world architecture design settings.

First, empirical studies with software architects and domain experts are required to assess whether the extended modeling constructs improve communication and shared understanding during system design. Controlled modeling experiments or practitioner workshops could compare traditional Context Mapper models with the extended DSL to evaluate differences in modeling accuracy, architectural transparency, and perceived usefulness.

Second, future work should investigate how the proposed constructs influence architectural decision-making in microservice decomposition processes. In particular, it would be valuable to analyze whether explicitly modeling Distillation Patterns and context relationships leads to more stable service boundaries and reduced architectural coupling over time.

Third, the DSL could be further extended toward tool-supported modeling environments. Integrating visual representations of the new patterns into Context Mapper and enabling automated transformations toward architectural artifacts (e.g., service contracts or architecture documentation) may strengthen the bridge between conceptual domain models and technical system design.

Finally, longitudinal studies in industrial environments would provide valuable insights into the scalability and long-term benefits of DSL-supported DDD modeling. Such studies could examine how domain models evolve over time and whether formalized modeling improves traceability and maintainability in large microservice landscapes.

References

Alshuqayran, N., Ali, N., & Evans, R. (2016). A Systematic Mapping Study in Microservice Architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)* (pp. 44–51). IEEE. https://doi.org/10.1109/SOCA.2016.15

Brandolini, A. (2020). Discovering Bounded Contexts with EventStorming. In DDD Community (Ed.), *Domain-Driven Design: The First 15 Years: Essays from the DDD Community* (pp. 37–57).

Brown, W. H., Malveau, R. C., McCormick III, H. W., & Mowbray, T. J. (1998). *Antipatterns: Refactoring software, architectures, and projects in crisis.* Wiley computer publishing.

Combemale, B., France, R., Jezequel, J.-M., Rumpe, B., Steel, J., & Vojtisek, D. (2017). *Engineering modeling languages: Turning domain knowledge into tools* (1st ed.). Chapman & Hall/CRC.

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software* (1. print). Addison-Wesley.

Evans, E. (2015). *Domain-Driven Design Reference: Definitions and Pattern Summaries.* Retrieved May 30, 2025 from https://www.domainlanguage.com/ddd/reference

Foote, B., & Yoder, J. (1999). *Big Ball of Mud*. http://www.laputan.org/mud/mud.html

Fowler, M., & Parsons, R. (2011). *Domain-specific languages*. Addison-Wesley.

Gartner Peer Community. (2023). *Microservices Architecture: Have Engineering Organizations Found Success?* Retrieved September 22, 2025 from https://www.gartner.com/peer-community/oneminuteinsights/omi-microservices-architecture-have-engineering-organizations-found-success-u6b

Grand View Research (2025). *Global Cloud Microservices Market Size & Outlook*. Retrieved September 24, 2025 from https://www.grandviewresearch.com/horizon/outlook/cloud-microservices-market-size/global

Kapferer, S., & Zimmermann, O. (2021). Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper. In S. Hammoudi, L. F. Pires, & B. Selić (Eds.), *Model-Driven Engineering and Software Development* (Communications in Computer and Information Science. Vol. 1361, pp. 250–272). Springer International Publishing. https://doi.org/10.1007/978-3-030-67445-8_11

Lewis, J., & Fowler, M. (2014). *Microservices - a definition of this new architectural term.* Retrieved May 21, 2025 from https://martinfowler.com/articles/microservices.html

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. A. (2021). Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology*, 131, 106449. https://doi.org/10.1016/j.infsof.2020.106449

Millett, S. (2020). Distilling DDD Into First Principles. In DDD Community (Ed.), *Domain-Driven Design: The First 15 Years: Essays from the DDD Community* (3–31).

Mohottige, T. I., Polyvyanyy, A., Fidge, C., Buyya, R., & Barros, A. (2025). Reengineering software systems into microservices: *State-of-the-art and future directions. Information and Software Technology*, 183, 107732. https://doi.org/10.1016/j.infsof.2025.107732

Newman, S. (2021). *Building Microservices: Designing fine-grained systems* (2nd ed.). O'Reilly.

Özkan, O., Babur, Ö., & van den Brand, M. (2025). Domain-Driven Design in software development: A systematic literature review on implementation, challenges, and effectiveness. *Journal of Systems and Software*, 112537. https://doi.org/10.1016/j.jss.2025.112537

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis N. (2017). Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software, 34*(1), 91–98. https://doi.org/10.1109/MS.2017.24

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008). Systematic Mapping Studies in Software Engineering, In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE)* (pp. 68–77). BCS.

Rademacher, F., Sachweh, S., & Zundorf, A. (2020). Deriving Microservice Code from Underspecified Domain Models Using DevOps-Enabled Modeling Languages and Model Transformations. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 229–236). IEEE. https://doi.org/10.1109/SEAA51224.2020.00047

Richardson, C. (2018). Microservices patterns: With examples in Java. Manning Publications.

Sangabriel-Alarcón, J., Ocharán-Hernández, J. O., Cortés-Verdín, K., & Limón, X. (2023). Domain-Driven Design for Microservices Architecture Systems Development: A Systematic Mapping Study. In *2023 11th International Conference in Software Engineering Research and Innovation (CONISOFT)* (pp. 25–34). IEEE. https://doi.org/10.1109/CONISOFT58849.2023.00014

Schmidt, R. A., & Thiry, M. (2020). Microservices identification strategies: A review focused on Model-Driven Engineering and Domain Driven Design approaches. In Á. Rocha (Ed.), *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)* (pp. 1–6). IEEE. https://doi.org/10.23919/CISTI49556.2020.9141150

Schmitz, Andreas; Pauken, Cedric; Kottmann, Renzo; and Wimmer, Maria A., "Architecture Interoperability through Semantic Modelling: Domain-driven data specification in public procurement" (2025). ECIS 2025 Proceedings. 9. https://aisel.aisnet.org/ecis2025/smart_gov/smart_gov/9

Taibi, D., & Lenarduzzi, V. (2018). On the Definition of Microservice Bad Smells. *IEEE Software*, 35(3), 56–62. https://doi.org/10.1109/MS.2018.2141031

Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.

Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182, 111061. https://doi.org/10.1016/j.jss.2021.111061

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14)* (pp. 1-10). ACM. https://doi.org/10.1145/2601248.2601268

Zhong, C., Li, S., Huang, H., Liu, X., Chen, Z., Zhang, Y., & Zhang, H. (2024). Domain-Driven Design for Microservices: An Evidence-Based Investigation. *IEEE Transactions on Software Engineering*, 50(6), 1425–1449. https://doi.org/10.1109/TSE.2024.3385835