INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA BIOMÉDICA

isep

Linguagem Específica de Domínio para a Configuração de Campanhas na Indústria Farmacêutica

PAULO MANUEL RIBEIRO ALMEIDA

outubro de 2025

P.PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

A Domain-Specific Language for Campaign Configuration in the Pharmaceutical Industry

Paulo Manuel Ribeiro Almeida

Dissertation submitted in partial fulfilment of the requirements for the degree of Master in Biomedical Engineering

Superviser: Nuno Silva

Co-superviser: Nuno Madureira

Júri:

Presidente: Prof. Dra. Natércia Lima, Professora-Adjunta, Instituto Superior de Engenharia do Porto

Vogais:

Prof. Dr. Nuno Silva, Professor-Adjunto, Instituto Superior de Engenharia do Porto

Prof. Dr. Paulo Maio, Professor-Adjunto, Instituto Superior de Engenharia do Porto

Porto, October 2025

Dedicatory

I dedicate this work to my family and friends, whose support and encouragement have been invaluable throughout the course of this dissertation.

Abstract

Campaign configuration in the pharmaceutical software is a complex and critical to business processes, where errors can directly affect revenue, compliance and customer satisfaction. In the ePharma platform, campaigns are currently embedded in the application code, making them rigid, error-prone and costly to maintain.The present work addresses these limitations by proposing, designing and implementing a Domain-Specific Language (DSL) and supporting rule engine to decouple campaign logic from the core system.

The DSL was designed to express business rules declaratively, in a human-readable format, to enable marketing and business analysts to define and modify campaign without requiring developer intervention.The implemented system adopts an input validation pipeline and a modular and extensible rule engine. The validation pipeline includes schema checks, semantic validation and runtime consistency checks.

The solution was validated through unit, mutation and integration tests, achieving 97% line coverage, 94% mutation coverage and full test strength. These results highlight the feasibility and robustness of the approach.

The outcomes suggest that this DSL-based approach is a sustainable solution for campaign management in pharmaceutical systems, and improved the maintainability, adaptability and agility of the previous system.

Graphical authoring tools and advanced validation are envisaged as future improvement opportunities.

Keywords: Domain-Specific Language, Rule Engine, Campaign Configuration, Pharmaceutical Industry, Business Rules, Software Architecture

Resumo

A configuração de campanhas comerciais em sistemas farmacêuticos representa um processo crítico, onde a precisão e a rapidez de resposta são essenciais para garantir a competitividade e a conformidade regulatória. Na plataforma ePharma, as campanhas estão atualmente definidas diretamente no código da aplicação, o que leva a uma arquitetura rígida, difícil de manter e altamente dependente de intervenção técnica. Este modelo resulta em custos extremamente elevados, lentidão na implementação e edição de campanhas e maior propensão a erros humanos.

Com o objetivo de ultrapassar estas limitações, este trabalho propôs, desenhou e implementou uma Linguagem Específica de Domínio (DSL), acompanhada por um motor de regras, destinada à configuração de campanhas. A DSL foi concebida com o intuito de permitir a expressão declarativa de regras de negócio num formato legível por humanos, promovendo assim a autonomia de analistas de negócio e equipas de marketing para criar, alterar e validar campanhas. Esta abordagem democratiza o processo de configuração, reduzindo a dependência em programadores e aumentado a agilidade na resposta às necessidades do mercado.

Um dos aspetos principais da solução é a pipeline de validação, que atua em diferentes níveis:

1. Validação estrutural, garantindo a correta utilização de campos e tipos.
2. Validação semântica, assegurando a consistência de parâmetros.

A implementação foi avaliada através de um conjunto abrangente de testes unitários, de mutação e de integração. Os resultados demonstram uma cobertura de código de 97%, confirmando a robustez da solução e a sua capacidade de deteção de falhas. A abordagem revelou claras melhorias em manutenibilidade, pela separação entre lógica de negócio e o código da aplicação; em adaptabilidade, através de uma arquitetura modular extensível; e em agilidade, pela possibilidade de definir e atualizar campanhas mais rápido e com mais segurança.

Os principais contributos desta dissertação podem ser classificados em três dimensões distintas:

- Conceptual – formalização do domínio da configuração de campanhas, incluindo modelos de entidades, especificações e restrições.
- Metodológica – definição de uma DSL, com serialização em JSON.
- Prática – implementação de um protótipo funcional integrado na plataforma ePharma, validado com testes sistemáticos e dados representativos.

Apesar dos avanços alcançados, algumas limitações foram identificadas. A utilização de JSON, embora legível, pode revelar-se complexa para utilizadores não técnicos ao expressar regras compostas profundas. A dependência de serviços externos para a construção do contexto de avaliação introduz pontos de falha, como riscos de disponibilidade e desempenho. Finalmente,

a manutenção contínua da pipeline de validação será necessária para acompanhar a evolução das regras de negócio.

Como trabalho futuro, três propostas foram identificadas:

- Ferramentas gráficas de autoria de campanhas – abstrair o JSON em componentes visuais (por exemplo, formulários e mecanismos de arrastar-e-largar).
- Mecanismos avançados de validação e simulação – permite observar o impacto antecipado das campanhas em diferentes cenários de dados com mais dinamismo.
- Otimização de desempenho e escalabilidade – explorar estratégias como cache e execução paralela para garantir tempos de resposta adequados mesmo em cenários de elevada complexidade.

Em forma de conclusão, este trabalho confirma que uma abordagem baseada em DSL constitui uma solução viável, sustentável e inovadora para o problema em questão, mais propriamente, a gestão de campanhas na indústria farmacêutica. A solução proposta responde às limitações identificadas, melhora a eficiência operacional e estabelece bases sólidas para futuras evoluções da plataforma ePharma.

Palavras-chave: Linguagem Específica de Domínio, Motor de regras, Configuração de campanhas, Indústria farmacêutica, Regras de negócio, Arquitetura de software

Acknowledgements

I would like to express my sincere gratitude to my family and friends for their constant support, patience and encouragement throughout this journey. Their understanding and motivation were essential in helping me complete this work.

I am also deeply thankful to my supervisors, Nuno Silva and Nuno Madureira. To professor Nuno Silva, I am forever grateful for all the teachings, his patience and his meticulous guidance, which were fundamental. To Nuno Madureira, I extend my thanks for his invaluable help, insightful opinions and the many long conversations that provided clarity and direction throughout this project.

I also wish to thank ISEP – Instituto Superior de Engenharia do Porto for providing the environment and resources that enabled the development of this work, the institution’s commitment to excellence and its supportive community played an essential role in making this dissertation possible.

My gratitude to ePharma, for giving me the opportunity to carry out this work in a real-world context. The company’s collaboration and shared expertise were imperative in ensuring that the project remained focused in practical challenges and industry needs.

Finally, I thank my colleagues and everyone who, directly or indirectly, provided their support, insights and encouragement during this process.

To all of you, I remain deeply grateful.

Content

Dedicatory iii

Abstract v

Resumo vii

Acknowledgements ix

List of Figures xiii

List of Tables xv

List of Code xvii

Introduction 1

1.1 Context 1

1.2 Problem Description 1

1.3 Objective 2

1.4 Approach 3

1.5 Contributions 4

1.6 Document Structure 5

State-of-the-art 6

1.7 Domain-Specific Languages (DSLs) – Concepts 6

1.8 Existing DSLs 8

1.9 DSL Creation tools 8
1.9.1 JetBrains MPS 9
1.9.2 Xtext 10
1.9.3 Internal DSL 11
1.9.4 Drools 12
1.9.5 Comparison 13

Analysis and Requirements 15

1.10 The ePharma System Architecture 15
1.10.1 Presentation Layer 16
1.10.2 Microservices Layer 17
1.10.3 Data Management Layer 18
1.10.4 Observability and Monitoring 19
1.10.5 DevOps and CI/CD 20

1.11 Current Campaign Configuration Process 20

1.12 Domain Analysis 24
1.12.1 Domain concepts 24
1.12.2 Campaign specification 25
1.12.3 Campaign’s Product Specification 26
1.12.4 Systematization 27

1.12.5 Core Domain Entities 28
1.13 Constraints 30

Design 33
1.14 Architecture 33
1.15 Core Design Principles 36
1.16 DSL 37
1.16.1 Syntax 39
1.16.2 Semantics 41
1.17 Summary 42

Implementation 45
1.18 DSL concrete syntax 45
1.19 Implementation Details 48
1.19.1 Rule Factory and Runtime Discovery 48
1.19.2 Rule Context and Evaluation 48
1.19.3 Intermediate Representation and Mapping from the DSL 49
1.19.4 Validation Pipeline 49
1.20 Benefits of the DSL Implementation 49
1.21 Limitations and Challenges 51
1.22 Comparison with Existing Approaches 51
1.22.1 Hard-coded business rules in application services 51
1.22.2 Configuration tables and database-driven rules 52
1.22.3 General-purpose Business Management Systems (BRMS) 52
1.22.4 Visual policy builders and low-code platforms 52

Verification and Validation 53
1.23 Unit Tests 53
1.23.1 Verification 53
1.23.2 Functional Verification 54
1.23.3 Coverage 54
1.24 Mutation Tests 55
1.25 Integration Tests 57

Conclusions 61
1.26 Summary of Contributions 61
1.27 Critical Reflection 61
1.28 Future Work 62
1.29 Final Remarks 62

Appendices 70

List of Figures

Figure 1 - Projectional editor example (JetBrains, 2025a)..9
Figure 2 - MPS quick fixes, code completion and intentions example (JetBrains, 2025). 10
Figure 3 - Xtext contextual code completion example (Xtext, 2025). 11
Figure 4 – Expression builder example (Fowler, 2010)..12
Figure 5 - Drools declarative logic example (Drools, 2013). 13
Figure 6 - ePharma system architecture...16
Figure 7 - Presentation layer. 17
Figure 8 - Microservices Layer. 18
Figure 9 - Data layer. 19
Figure 10 - Orders functionality class diagram. 25
Figure 11 – Analysis domain diagram. 29
Figure 12 - Activity diagram of campaign authoring and evaluation............................34
Figure 13 - Evaluation sequence diagram. 35
Figure 14 – DSL class diagram..37
Figure 15 - Abstract Syntax Tree of a campaign example. 40

List of Tables

Table 1 - DSL creation tools comparison. 13
Table 2 - Unit tests...54
Table 3 - Mutation tests results..56
Table 4 - Mutation tests class results. 56
Table 5 - Integration tests. 58

List of Code

Code 1 - Execution order code example. 21
Code 2 - Campaign document example. 23
Code 3 - DSL concrete syntax example. 46
Code 4 - Atomic rule concrete syntax example. 47
Code 5 - Composite rule concrete syntax example. 47
Code 6 - Nested rules concrete syntax example. 48
Code 7 - Unit test example 54
Code 8 – Functional verification unit test example. 54
Code 9 - Integration test DSL example. 57
Code 10 - Integration test example. 58

List of Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
BRMS	Business Rule Management System
CI/CD	Continuous Integration / Continuous Deployment
CSS	Cascading Style Sheets
DBMS	Database Management System
DSL	Domain-Specific Language
FURPS+	Functionality, Usability, Reliability, Performance, Supportability +
HTML	HyperText Markup Language
DSL	Domain-Specific Language
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
ISO/IEC	International Organization for Standardization / International Electrotechnical Commission
JSON	JavaScript Object Notation
JWT	JSON Web Token
LINQ	Language Integrated Query
PIT	Pitest
REST	Representational State Transfer
SFTP	Secure File Transfer Protocol
SQL	Structured Query Language
UML	Unified Modeling Language
XML	eXtensible Markup Language
YAML	Yet Another Markup Language

Introduction

1.1 Context

Based in Portugal, ePharma is a technology company that has been operating in the pharmaceutical software for over 20 years. It develops solutions for companies in this sector to help in their processes, through the configuration and adaptation of different modules. With ePharma, companies can easily manage their marketing and sales activities with personalized developments and integration with pharmaceutical distribution networks. The company's primary objective is to enhance the efficiency and accuracy of business processes in this highly regulated and competitive domain (ePharma, 2024).

The product, also named ePharma, is a platform that allows the clients – primarily pharmaceutical laboratories, distributors and pharmacies – to manage orders, commercial campaigns, business policies and others, with high granularity. A key feature in the ePharma platform is the Orders, and it represents a critical aspect of the system's value which enables clients to manage complex pharmaceutical order processes with precision and flexibility. This feature supports the creation and submission of structured orders, tailored to client-specific needs, allowing for the coordination of multiple products, suppliers and delivery schedules within a single order flow.

Orders are tightly integrated with the platform's campaign system, which dynamically applies business rules such as discounts and bonuses based on the contents and context of each order. This integration is imperative to ensure that all relevant commercial conditions are met, reinforcing the importance of the Orders module in business rule enforcement and client-specific sales conditions.

These campaigns are a critical aspect of the system's business value, enabling pharmaceutical companies to implement flexible campaigns that can be customized per client, meeting different needs gracefully. Typically, this includes conditional logic based on quantities, value or customer profile attributes. For example, a campaign – or commercial policy – might specify that purchasing 100 units of a product, entitles the buyer to a 15% discount and five bonus units, but only if the requester has not exceeded a maximum allowed purchase volume or meets certain specific conditions.

The current implementation of the ePharma system follows a microservices architecture, developed using Java with Spring Boot. The backend services are containerized and orchestrated to promote scalability, maintainability and modular evolution of new features. Data storage is handled by a MySQL and MongoDB database, hosted on Microsoft Azure.

1.2 Problem Description

In enterprise applications, business rules often evolve rapidly and may differ across domains. Hardcoding these rules into the core business logic of the application results in a brittle system, that is difficult to maintain and extend. In the ePharma application, the campaigns, used in the Orders functionality, relies on business logic to work properly. Campaigns have very different

logic, depending on the client, scope and many other variables. This makes this part of the system very volatile, with the need for constant updates.

Despite the state-of-the-art technologies, parts of the system are still outdated or overly rigid, making maintenance more complex and limiting the ability to adapt and evolve to new business requirements. As the company grows, these limitations become more apparent, highlighting the need for a more flexible and scalable system, to keep up with the market demands.

The configuration of campaigns in the ePharma platform is a manual and outdated process, that requires developers to directly modify the core business logic whenever new campaigns and updates are requested by the clients. This makes this approach time consuming and highly prone to errors, where even small adjustments can introduce unintended inconsistencies.

In the pharmaceutical industry, new campaigns are frequently created and updated, so the current approach imposes a significant difficulty in maintenance. Changes such as product discount, bonus quantities and other promotional logic have an urgent need to be updated in the shortest possible time, so the campaigns reflect these changes to the end user. Since these alterations are made manually and they lack a more streamlined and intuitive configuration mechanism, the process is very slow which affects the company's ability to quickly respond to market demands.

ePharma has a substantial number of clients, each with very specific needs. The current system lacks the needed flexibility to adapt to client-specific needs, as the configuration process is tightly coupled to the core business logic. This inflexibility limits the system's ability to quickly apply custom business logic for different clients or business requirements.

The system contains legacy design patterns, inherited from earlier versions of the product. Over time, the accumulation of inconsistent coding practices and technical debt – particularly the configuration and maintenance of campaigns – have introduced some limitations to the growth of the system.

The current setup relies on internal procedures to implement the requested changes made by the clients, that is dependent on several human factors – such as interpretation and oversights – that increases the likelihood of errors during configuration and impacts business operations and client satisfaction. This means that every new request, even a small change, involves code alteration, testing and redeployment, often with little automation or safeguards. This approach is not only time-consuming and error-prone but also imposes several limitations in terms of configurability, adaptability and extensibility of the system, especially as the number of clients grow and, consequently, the number of campaign variants. This growth as highlighted these limitations, as clients demand for faster response times and real time configuration capabilities to respond to volatile market demands. The inability to accommodate the dynamic requirements with agility undermines both operational efficiency and customer satisfaction. The current solution's inefficiencies highlight the urgent need for a more sustainable, resilient and modern approach to configure campaigns.

1.3 Objective

The main objective of this thesis is to modernize ePharma's campaign configuration system by implementing a more efficient process. The goal is to idealize and develop a more flexible,

adaptable, and maintainable solution that improves the speed and accuracy of campaign updates, while reducing the dependency on developer intervention.

More specifically, this thesis pursues the following objectives:

- Idealize a dedicated campaign configuration system, enabling the definition of rules and constraints in a formal yet accessible manner by the business analysts or marketing teams. Provide business analysts and marketing teams with the ability to directly compose, modify and manage campaign logic through a declarative, human-readable syntax, minimizing reliance on software developers.

- Ensure the system can seamlessly accommodate evolving business requirements, client-specific needs and changing conditions without requiring extensive development efforts.

- Incorporate mechanisms for both design-time and runtime validation of rules, thereby reducing errors and ensuring both correctness and consistency of the campaign configurations.

- Establish a foundation that simplifies integration, testing and future extension of the system, ensuring that it remains robust and adaptable as the business evolves.

This initiative has the objective of empowering non-technical stakeholders – such as business analysts and marketing teams – to be able to compose and manage campaigns in a declarative and human-readable format.

1.4 Approach

To properly address the complexity of the configuration process, the development of a Domain-Specific Language (DSL) with a supporting rule engine was proposed. By adopting this new paradigm, the goal is to improve maintainability, reduce the time needed to update or create campaigns and support a broader range of client-specific needs through a more extensible, modular and user-friendly architecture.

The envisioned DSL has been designed to abstract away the technical intricacies of campaign logic specification, enabling business experts and technical teams to interact seamlessly while ensuring the correctness and efficiency to the processes. Complementing the DSL, the rule engine provides the execution environment in which authored rules are dynamically registered, composed and evaluated. By introducing a high-level, declarative approach, the DSL provides a more intuitive and error-resistant mechanism for defining campaign constraints and business rules, thus minimizing the impact of human factors and improving the accuracy of campaign authoring and execution.

A core objective of this DSL is to facilitate the quasi-automatic integration of different campaign entities, services and data sources. Instead of requiring to manual work or rigid configurations, the DSL allows rules and constraints to be expressed declaratively and consistently, while the rule engine enforces them in real time against the operational context enabling rigorous methods of validation, customization and deployment. Design-time validation mechanisms

further guarantee early detection of errors such as structural inconsistencies, missing parameters or logical contradictions, while run-time validation ensures that rules remain consistent with dynamic business data.

Moreover, the DSL syntax has been designed with usability in mind. Features such as a custom user interface will help users in constructing campaigns that are both syntactically and semantically valid. This combination of abstraction, validation and automations ultimately enables the platform to accommodate new business rules, client-specific requirements and evolving market demands more rapidly and reliably, while significantly reducing the dependency on manual intervention.

Finally, introducing a DSL supported by a dedicated rule engine will enable a more error-resistant and maintainable method for defining and applying campaign logic. This transition would improve the system's efficiency, flexibility and adaptability, ultimately enhancing the quality of the product and reducing costs.

1.5 Contributions

This thesis makes several key contributions to the modernization of ePharma's campaign configuration system by introducing a Domain-Specific Language (DSL) specifically designed for the definition, validation and execution of campaign rules. These contributions can be grouped into conceptual, methodological and practical outcomes.

First, at the conceptual level, the work formalizes the domain of pharmaceutical campaign configuration, which previously lacked a systematic representation. Through extensive analysis and abstraction, a domain model was developed to capture the core entities, relationships and constraints governing the configuration process. This domain model serves as a foundation upon which the DSL was designed, ensuring that the language faithfully reflects the semantics of the business domain.

Second, at the methodological level, this thesis introduces a DSL tailored to the needs of campaign configuration. The DSL provides both an abstract syntax – capturing the essential constructs and their relationships – and a concrete syntax, designed for practical usability and integration with the ePharma system. By separating these layers, the DSL achieves a balance between expressive power and user accessibility, enabling business experts and developers to collaborate in a more efficient way, without being constrained by low-level technical details.

Third, this work conceptually contributes an architecture and execution framework that supports dynamic rule registration, composition and evaluation. This framework incorporates a validation component that operates both at campaign design-time (detecting syntactic and contextual errors early) and at run-time (ensuring the consistency of rule execution with dynamic business data). This dual validation approach significantly reduces the likelihood of configuration errors and enhances the reliability of campaign execution.

Finally, at a practical level, this thesis delivers a working prototype integrated into the ePharma platform, demonstrating the applicability and benefits of the proposed DSL. The implementation showcases how the DSL improves the speed, accuracy and maintainability of campaign authoring, while enabling adaptability to new business requirements and client-specific needs. By reducing dependency on manual configuration and technical intervention,

the DSL provides a sustainable and future-proof solution to campaign management. The presented project is quite challenging and would make a positive impact in the way that campaigns are being implemented, employing widely adopted technologies by major corporations that seek to improve the quality of their products. This transition would not solely reduce the company's expenses but also facilitate code comprehension and modification, following the best programming guidelines.

1.6 Document Structure

The remainder of this dissertation is organised into 7 chapters, each addressing a different aspect of the research.

Chapter 2 presents the state of the art, introducing the fundamental concepts of Domain-Specific Languages (DSL), surveying existing DSLs and creation tools.

Chapter 3 focuses on the analysis of the ePharma system, with particular attention to the campaign configuration process, its limitations and the modelling of the core domain entities that serve as the conceptual foundation for the redesign proposed. It also specifies the constraints of the solution, both functional and non-functional, establishing the criteria against which the proposed solution was designed and later evaluated.

Chapter 4 discusses the design of the solution, outlining the core design principles, architectural overview and the syntax and semantics of the proposed DSL.

Chapter 5 addresses the implementation, detailing how the DSL and the supporting rules engine were developed and integrated into the ePharma platform, while also highlighting benefits, limitations and the comparison with existing approaches.

Chapter 6 covers the verification and validation, covering the testing strategy adopted, including unit, mutation and integration tests.

Finally, Chapter 7 concludes the dissertation by reflecting on the main contributions, the results achieved and identifying directions for future work.

State-of-the-art

In this chapter, a focus will be given to the concept of Domain-Specific Languages (DSLs) and existing works related to the application of DSL's will be explored.

1.7 Domain-Specific Languages (DSLs) – Concepts

Domain-Specific Languages are widely used in many different business applications due to their ability to simplify complex processes and enhance maintainability. Companies often opt to create a DSL because the languages are compact and tailor-made for their specific use cases, resulting in a more error-resistant, well-structured and flexible solution (TLVTech, 2025). Domain-Specific languages are computed languages tailored to specific business needs that offer expressiveness and ease of use, compared to common programming languages (Mernik, Heering and Sloane, 2005). They enable users to create a language that solves a specific problem in a particular domain more naturally and efficiently by encapsulating the domain knowledge (Hermans, Pinzger and van Deursen, 2009). DSLs present various advantages over General-purpose programming languages (Kosar et al., 2010) and has been described in literature for several decades.

DSLs have a focused scope, they are crafted to address problems within a specific domain, providing constructs that directly map to domain concepts (Mernik, Heering and Sloane, 2005). By abstracting complex operations into simpler domain-relevant terms, DSLs enhance productivity as they streamline the development process, reducing potential errors and improving maintainability (Berzak, 2023). DSLs are designed to match the format typically used by the domain experts, such that they can directly specify, implement and validate without the need of coding intermediaries (Spinellis, 2001). According to Martin Fowler, DSLs may improve communication with the domain experts, which is one of the hardest problems in software development (Fowler and Parsons, 2010).

Martin Fowler also separates DSLs into two main forms: internal and external. Internal DSLs are embedded within a host general-purpose language, leveraging its syntax and semantics. External DSLs on the other hand, are stand-alone languages, with their own syntax and parsing mechanisms, independent of any host language (Artho et al., 2015). Neal Ford considers LINQ (Language Integrated Query) as a good example of an internal DSL, because "the LINQ syntax you use is legal C# syntax, but an extended (domain specific) syntax.". With this line of thought, JUnit can also be considered an internal DSL, since it exploits the host language's features to provide domain-specific expressiveness. External DSLs, on the other hand, can include SQL, HTML and LaTeX, because they have their own syntax and parsing mechanisms. (Quality Management, 2018)

While Domain-Specific Languages offer significant benefits, they also present certain challenges. Creating a DSL can be resource-intensive, requiring both specialize knowledge in language design and in the domain where the DSL will be used (Gray et al., 2008). The initial investment may be substantial, since the cost and time involved in the development is high and in smaller organizations, where resources are limited, it may not be justifiable (Sharma, 2024). Even though DSLs are intended to be accessible to domain experts, they do introduce a new

language that the users must learn. This learning process can be difficult, especially if the DSL syntax or ideas are complicated (TLVTech, 2025).

Federico Tomassetti and Vadim Zaytsev, highlight some of the adoption problems of DSLs, mentioning some professionals are unaware or find them risky. They find that when there is a proper understanding of DSLs within an organization, the perceived risks are the lack of competence and adoption (Zaytsev and Tomassetti, 2020).

DSLs present some challenges regarding he high initial development costs, steep learning curve, and potential adoption problems (Borum, 2022), but, in some cases, the long-term benefits might outweigh these limitations. They have the capability to enhance productivity, reduce errors (Christensen, 2003) and provide clear communication between the domain experts and the developers, making them invaluable in certain fields (Réveillere et al., 2000). Industries are starting to recognize the value of tailored solutions, like the adoption of DSLs, leading to further advancements in software development and domain-specific problem-solving (Paczona et al., 2024).

So, the advantages and disadvantages of DSL can be summarized as:

Advantages:

- Simplify complex processes and enhance maintainability.
- Compact and tailor-made for specific use cases, making them error-resistant, well-structured and flexible.
- Provide expressiveness and ease of use compared to general-purpose languages.
- Directly map constructs to domain concepts, abstracting complex operations into simpler domain-relevant terms.
- Enhance productivity, streamline development, reduce potential errors and improve maintainability and adaptability.
- Match the format used by domain experts, enabling them to specify, implement and validate.
- Improve communication with domain experts, one of the hardest problems in software development.
- Increasingly recognized by industries as valuable solutions, leading to advancements in domain-specific problem solving.

Disadvantages:

- Resource-intensive to create, requiring specialized knowledge in both language design and the target domain.
- High initial investment in cost and time, which may not be justifiable in smaller organisations.

- Introduce a new language that users must learn, which can be difficult if the syntax or concepts are complex.

- Adoption problems due to the lack of awareness, perceived risks or lack of competence within organisations.

- Steep learning curve and potential adoption barriers.

1.8 Existing DSLs

Domain-Specific Languages have gained substantial traction across many industries, for its effectiveness in encapsulating business logic, improving maintainability and empowering domain experts. Over the years, several widely used DSLs have emerged, each designed to address the needs of a particular sector or business type.

One of the most recognized DSL is SQL, designed to manage and query relational databases, it's a tool for communicating with the Database Management System (DBMS) (Desai, Hoffman and Kopalová, 2014). It has a declarative syntax that allows users to describe what data they want, without the need to describe how to retrieve it.

Hibernate Query Language (HQL) is another widely used DSL that allows the mapping of Java classes into tables of a relational database, providing the ability to write queries in a form similar to SQL but in terms of Java classes (Fowler, 2010). This is a strong example of how a DSL makes users think of business terms instead of technical ones.

Another very popular DSL is HTML (HyperText Markup Language), widely used in the web domain to describe the structure and presentation of information in web pages, indicating how the browser should structure or present the document (Powell, 2010). Despite its simplicity in terms of syntax, HTML is a powerful language when combined with other technologies, for example, CSS, another very powerful DSL.

These examples illustrate the diversity of DSLs and how each has been optimized for its respective domain. Their success comes from their ability to express domain logic succinctly, reduce cognitive load and to facilitate automation.

1.9 DSL Creation tools

In this section, several options for Domain-Specific Language creation were evaluated. Given the ePharma application is built using Java and the Spring Framework, the main goal was to find an approach that integrates smoothly into de existing codebase, minimizing dependencies. The focus was on two well-established external DSL creation tools – JetBrains MPS and Xtext- and an internal DSL approach, built directly with Java.

1.9.1 JetBrains MPS

JetBrains MPS (Meta Programming System) is a language workbench developed by JetBrains that represents code structure visually, without relying on traditional parsing (JetBrains, 2025). Instead of writing code in plain text, JetBrains MPS maintains programs as abstract syntax trees. This approach eliminates the need for traditional parsing and ensures that every edit corresponds to a syntactically correct model (JetBrains, 2025; Fowler, 2014). As a result, users manipulate structured models that are visually rendered in code-like notations but are in fact stored and managed as semantic structures rather than plain text. This enables MPS to support textual, graphical, or tabular notations over the same underlying model (Voelter and Lisson, 2014).

For example, Voelter (2007) describes how MPS's modular and compositional language design facilitates rigorous static analyses, automated propagation of changes, and enforcement of consistency rules across complex artefacts. Similarly, Dusan et al. (2014) reported that when using MPS to implement a requirements specification language, the structural editing and validation mechanisms helped ensure correctness and maintainability of specifications in industrial settings.

It uses a projectional editor that allows users to edit the Abstract Syntax Tree in an efficient way. The user can interact with the code through intuitive visuals, like tables and graphical languages (JetBrains, 2025a). An example of projectional editor is shown in Figure 1.

```
System.out.println(String.valueOf((∑ [[1 k 0] [0 1.0 0] [0 0 1]])));
System.out.println(exp(a + i * b) - exp(a) * (cos(b) + i * sin(b)));
matrix<Double> s = ...;
```

Figure 1 - Projectional editor example (JetBrains, 2025a).

MPS provides extensive code-completion features, offering intelligent suggestions for constructs and language concepts as shown in Figure 2. The syntax highlighting is provided through protectional editing, which visually differentiates constructs. The error diagnostics is a strong component of MPS, offering real-time validation and type system checking. When it comes to refactoring, this tool provides a comprehensive support, including renaming, restructuring and automated propagation of changes (JetBrains, 2024). All these features come together with a tight integration with JetBrains IDE's, supporting navigation, version control and collaborative work.

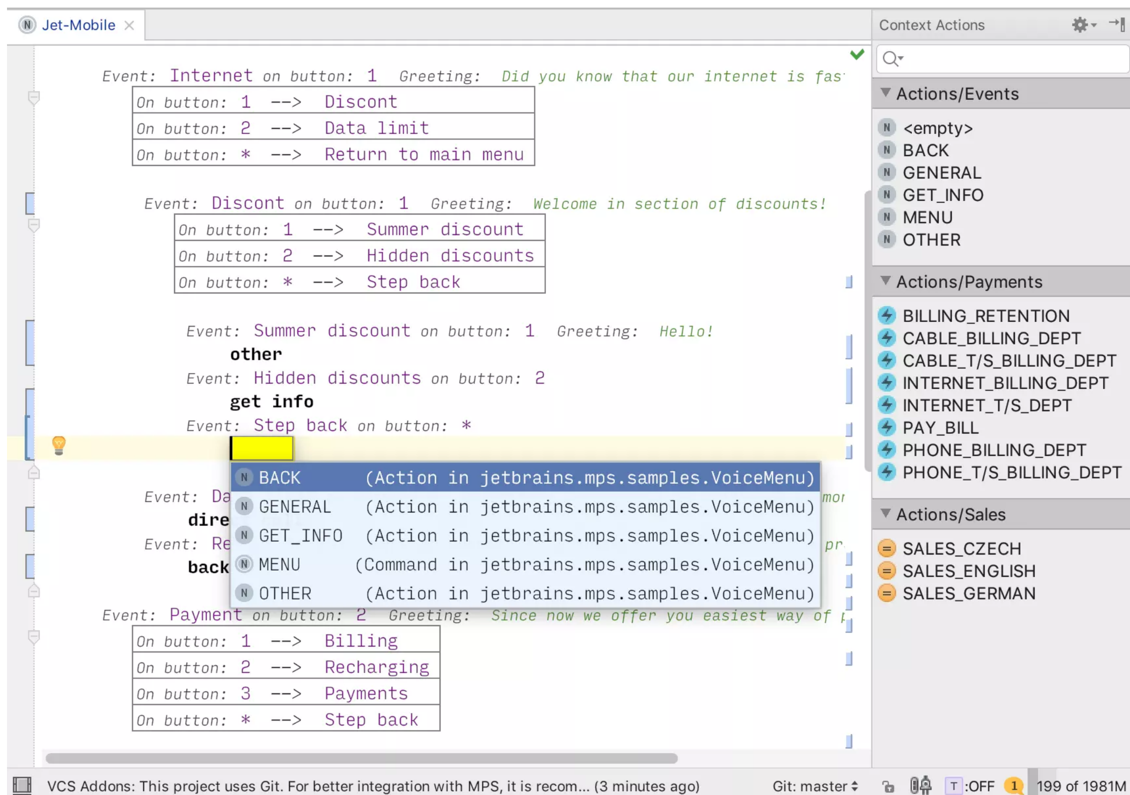

Figure 2 - MPS quick fixes, code completion and intentions example (JetBrains, 2025).

However, MPS comes with a steep learning curve and imposes significant change in development workflow (Voelter et al., 2010). Moreover, because it operates outside the traditional Java ecosystem, even though it is possible integrating its output into a Spring-based application, would add complexity, requiring bridges between generated code and the Spring context. For instance, Koščejev (2022) shows how generated Java classes must be loaded from an MPS module's classloader and invoked reflectively. This externality can make maintainability and collaboration more difficult, particularly in team that are not familiar with model-driven development paradigms.

1.9.2 Xtext

Xtext is another mature framework aimed at defining textual DSLs using a formal grammar. Based on the Eclipse Modeling Framework, Xtext allows developers to specify a language's syntax in a concise manner and automatically generates a parser, semantic model and a rich editor with syntax highlighting and validation features (Xtext, 2025).

In terms of code completion, Xtext offers contextual code-completion for DSL constructs, provided by the built-in editor. The syntax highlighting is automatically generated for grammar-defined tokens with a real-time grammar and semantic validation with editor feedback. The editor's refactoring is basic, supporting renaming grammar rules and updating references. It is deeply integrated into the Eclipse ecosystem, generating eclipse plug-ins with full editor support (Schneller, 2010). Figure 3 shows an example of some of the capabilities of Xtext.

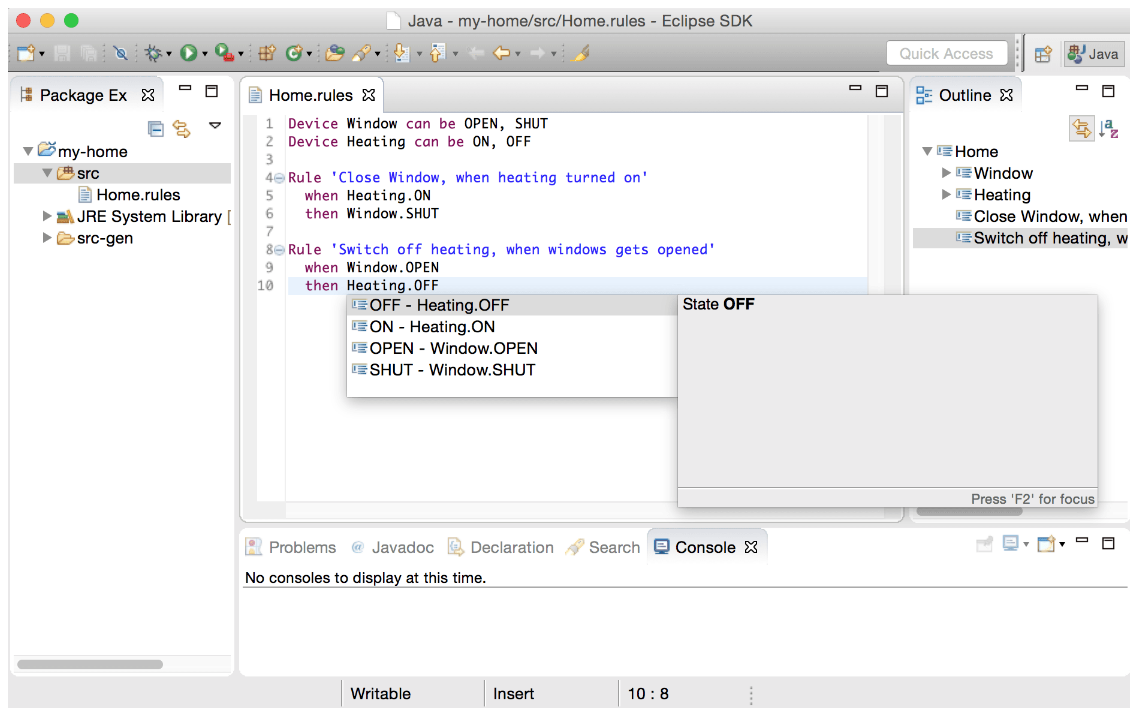

Figure 3 - Xtext contextual code completion example (Xtext, 2025).

However, its dependency on the Eclipse ecosystem makes it less attractive in a team that primarily uses another development environments such as IntelliJ IDEA. The domain in question demands validation of a range of non-functional concerns, including timing safety and fault tolerance. The default validation and processing behaviour provided by Xtext is insufficient to fully address these requirements. To bridge this gap, a customization of Xtext's behaviour would be needed, particularly in how models and expressions are processed. This customization typically relies on the use of Xtend, a language designed to complement Xtext, by enabling more expressive model transformations and logic (Xtext, 2025).

As such, effective use of Xtext in this context would require not only a solid understanding of its grammar and tooling but also proficiency in Xtend. This dual dependency implies higher onboarding complexity, increased development effort and potentially elevated costs. While using Xtext and Xtend in combination is technically feasible, the additional burden in terms of learning curve and integration effort renders this approach impractical for the needs of the present solution (Xtend, 2025).

1.9.3 Internal DSL

The Internal DSL approach leverages the expressiveness of the host language, in this case, Java – to define domain-specific constructs within the application itself (Pawar, 2025). Internal constructs can be tightly integrated with Spring components such as services, repositories and configurations, allowing the DSL to directly invoke business logic without translation layer or inter-process communication (Ruiz and Bay, 2008).

Fowler emphasizes that internal DSLs often exploit the flexibility of the host language's syntax that often involves method chaining, builder patterns and carefully chosen naming conventions

to make the code more business-intuitive and self-documenting. In Java, this can be achieved by designing APIs that guide developers through the correct sequence of method calls, reducing errors and improving readability (Fowler, 2006). Figure 4 is an example of an expression builder, by Martin Fowler.

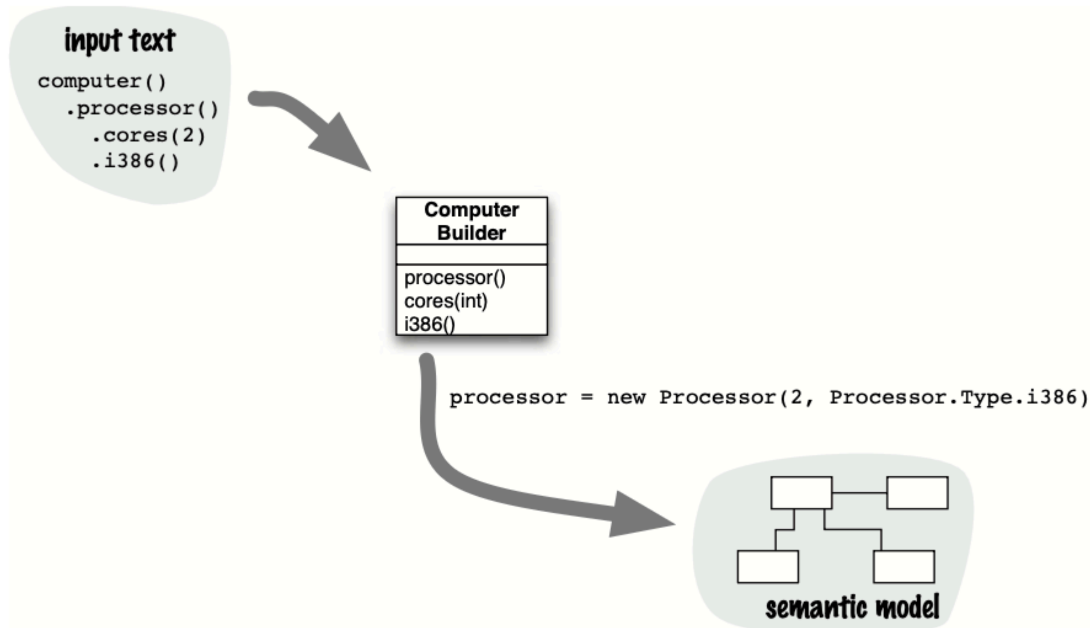

Figure 4 – Expression builder example (Fowler, 2010).

The code completion and syntax highlighting features are indirectly supported through the Java IDEs, where fluent APIs and method chaining can benefit from the built-in IDE autocompletion. When it comes to error diagnostics, it is fully supported by the Java compiler and IDEs, with real-time error feedback. The refactoring is a strong feature of Internal DSLs, since IDEs provide extensive refactoring tools for renaming, extracting and moving methods/classes. Another strong feature is the seamless integration with the Java ecosystem, including Spring, Maven/Gradle and various testing frameworks.

By building the DSL internally, the team can keep all the development within a unified codebase. In this approach, there is no need for external DSL editor, custom parsers or separate runtime environments (Fowler, 2008). This approach also benefits from immediate compatibility with Spring features like dependency injection, lifecycle management and transactional behaviour.

While internal DSLs may lack some of the syntactic freedom and sophisticated grammar definition tools of external DSLs, the trade-off is worthwhile in scenarios where seamless integration, simplicity and maintainability are more critical than language flexibility (Fowler, 2005).

1.9.4 Drools

Drools is also a very popular Business Rule Management System (BRMS) that provides a rich DSL for defining and managing business logic through rules (Drools, 2025). It uses Drools Rule Language (DRL), that allows business analysts to encode logic declaratively, similar to the

objectives of the DSL proposed in this thesis. Drools demonstrates how rule engines can support dynamic decision-making in enterprise systems (Kumar et al., 2011). Figure 5 exemplifies these functionalities.

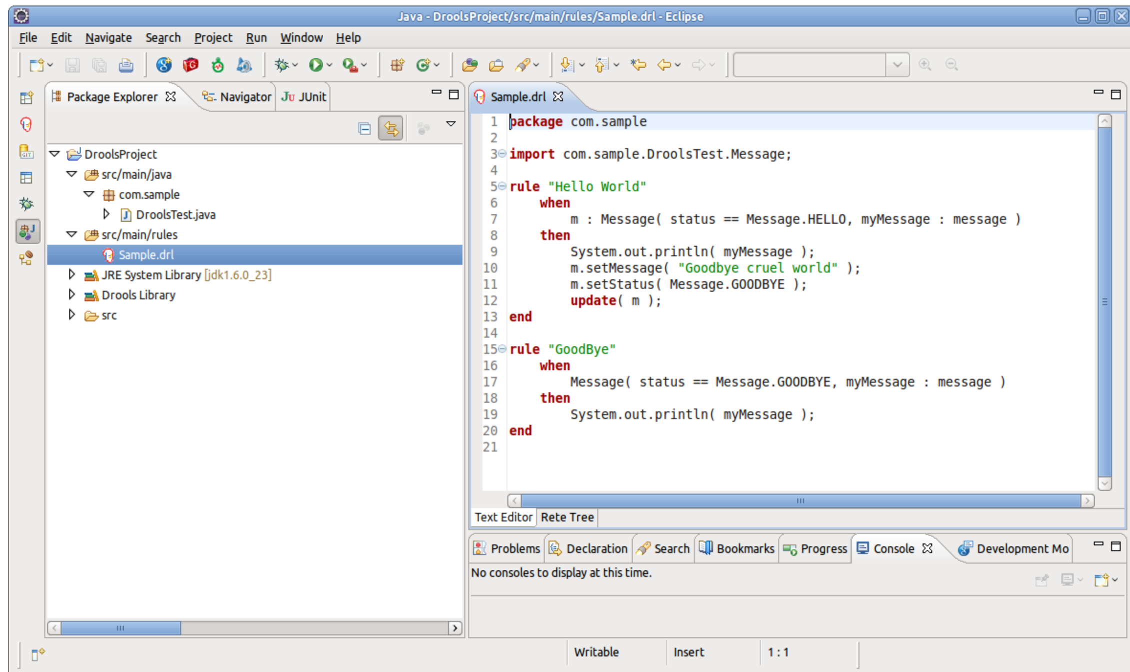

Figure 5 - Drools declarative logic example (Drools, 2013).

Drools code completion can be achieved with the use of IDE plugins, like in Eclipse and IntelliJ IDE, also offering other features like content assistance and syntax highlighting when editing .drl and .dslr files. Error diagnostics also comes with the use of an IDE with a Drools plugin, offering real-time validation and error markers for incorrect or inconsistent rule constructs. The refactoring when using this BRMS is a weak spot, since it has a limited automated refactoring and is not fully supported.

Drools is a widely used platform, used in multiple different domains, such as banking and financial services, insurance companies and telecommunications. Companies like Jpmorgan Chase & Co., American Express Company, Cisco Systems, Inc. and Visa Inc., are just some examples of corporations that use this platform for their business specific activities (HGData, 2025).

Although Drools provides strong rule-management features, its limited support and reliance on plugins can be restrictive compared to more flexible options.

1.9.5 Comparison

The following table provides a comparative overview of the four approaches:

Table 1 - DSL creation tools comparison.

Characteristics/Feature	MPS	Xtext	Drools	Internal DSL

Code Completion	Yes (projectional)	Yes (grammar based)	Yes (via Java IDEs with plugins)	Yes (via Java IDEs with plugins)
Syntax Highlighting	Yes (projectional)	Yes	Yes (via IDEs with plugins)	Yes (via Java IDEs with plugins)
Error Diagnostics	Yes, (type system checks)	Yes (real time)	Yes (real-time)	Yes (Java compiler)
Refactoring Support	Strong (rename, move)	Basic (rename)	Limited (manual in most cases)	Strong (IDE-supported)
Tool Integration	JetBrains IDE ecosystem	Eclipse ecosystem	Java IDEs with plugins	Native Java + Spring

Analysis and Requirements

1.10The ePharma System Architecture

The ePharma platform is implemented as a cloud-native microservices system, deployed on Microsoft Azure. This technology stack is robust and widely used, known for its scalability, modularity and the strong support for enterprise-grade microservices (Swift, 2024). At the highest level the architecture comprises:

- Customers and stakeholders access the application via web, mobile, API and SFTP.
- API Gateway that centralises ingress.
- Set of independently deployable Spring-based microservices that implement domain capabilities (Orders, Products, Entities, etc.).
- Asynchronous messaging (Azure Service Bus) used for event propagation and system decoupling.
- Polyglot persistence layer with a primary relation database (MySQL), a document store (MongoDB), an in-memory cache (Redis) and archival object storage (Azure Blob Storage).
- Platform services for secrets (Key Vault), observability (Application Insights / Azure Monitor / Log Analytics) and DevOps (Azure DevOps).

Figure 6 shows the current architecture of the ePharma platform.

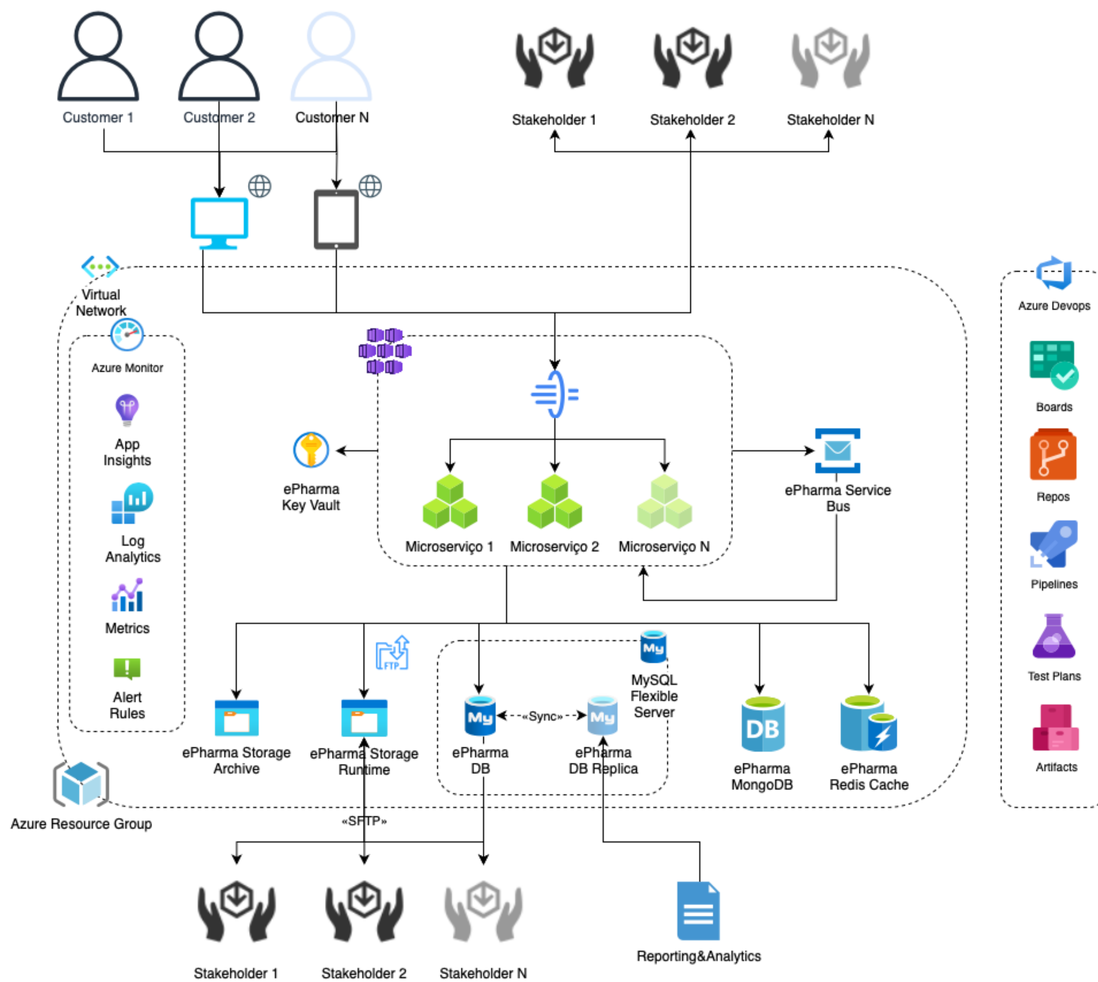

Figure 6 - ePharma system architecture.

1.10.1 Presentation Layer

At the top of the architecture is the presentation layer, shown in Figure 7Figure 7, comprising the ePharma App and Portal. These user interfaces serve customers, pharmacists, distributors and internal administrators. Implemented using Angular, they deliver responsive web applications accessible across browsers and devices.

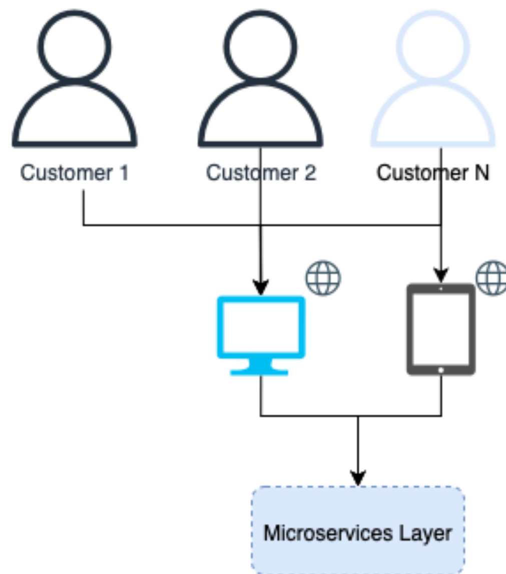

Figure 7 - Presentation layer.

The API Gateway acts as a single-entry point to backend services, in the microservices layer. It is responsible for directing client requests to the correct microservice and handling authentication and authorization using OAuth2 and JWT.

This simplifies client communication and improves resilience by decoupling clients from internal service topology.

1.10.2 Microservices Layer

The microservices layer forms the system's core (Figure 8). Developed in Java with Spring Boot and deployed in containers, each microservice addresses a bounded business domain. For example, Orders, Campaigns, Products and Entities. These services communicate via REST/HTTPS requests and via Azure Service Bus in some use cases.

All client interactions pass through the API Gateway, ensuring security, abstraction and uniform access to backend services.

The ePharma Service Bus facilitates asynchronous integration across services. It supports publish-subscribe communication patterns, ensuring reliable delivery and decoupling between producers and consumers. For example, campaign publication events are propagated to dependent services, enabling eventual consistency without blocking user-facing operations.

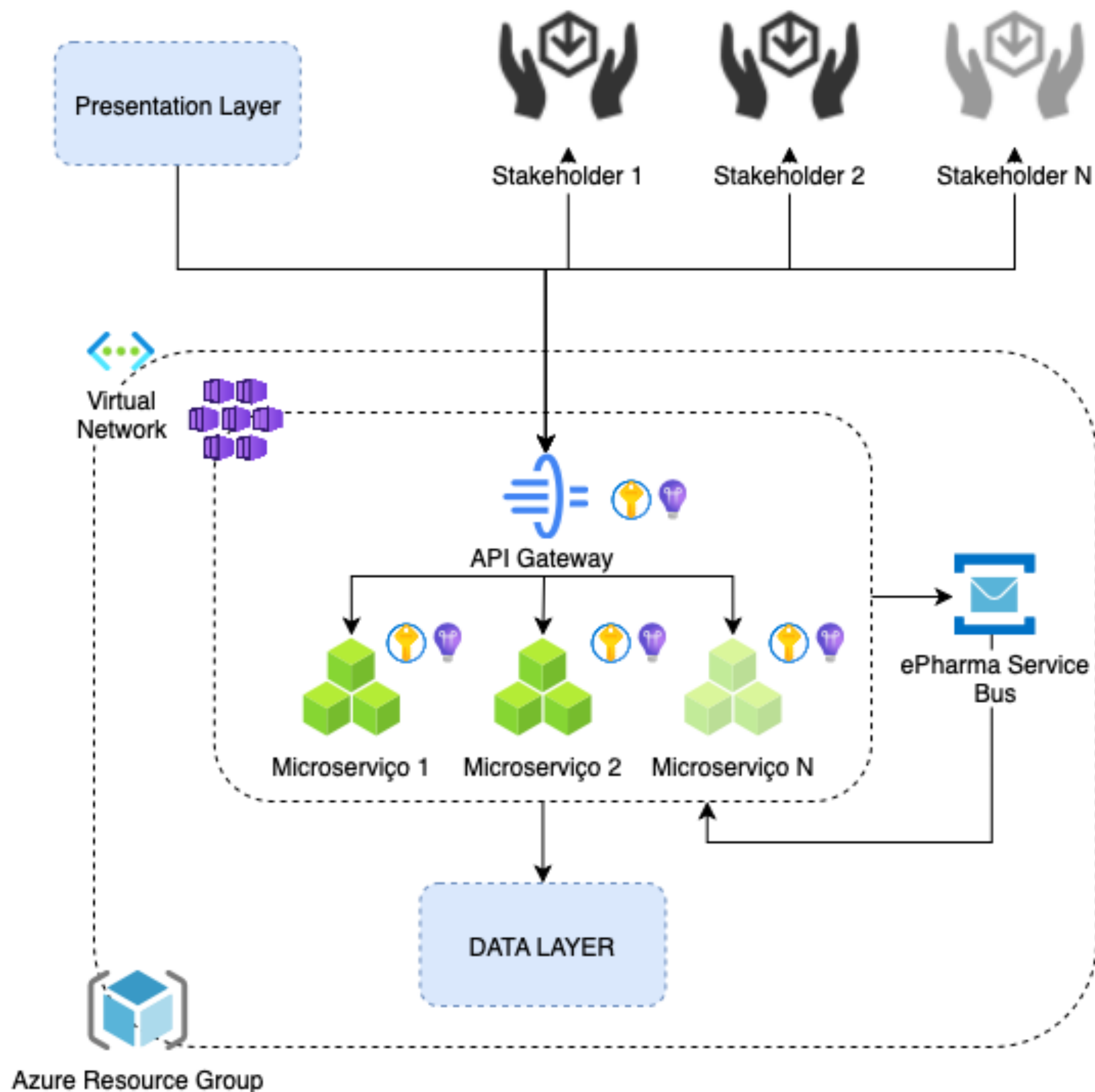

Figure 8 - Microservices Layer.

This structure promotes isolation and resilience, which enables independent development, deployment and scaling, and fault tolerance by isolating failures within a single service. This modularity and adaptability, enables for continuous delivery of new features.

1.10.3 Data Management Layer

The data layer is represented in Figure 9Figure 9 and it integrates diverse persistence technologies:

- MySQL Flexible Server: Central relational store for transactional data (orders, payments, users).
- Database Replicas: Read-only replicas improve scalability and support reporting.

- MongoDB: Document store for campaigns.
- Redis Cache: Provides low-latency access to compiled rules, pricing data and other frequently accessed objects.
- Blob Storage: Retains files and historical records with lifecycle policies and SFTP access for stakeholders.

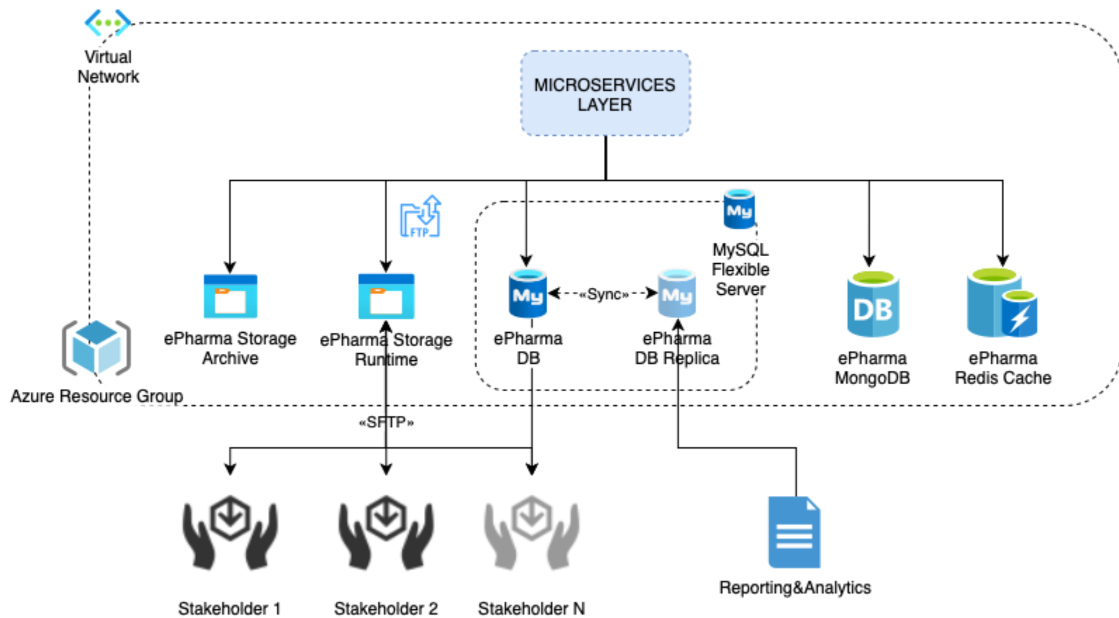

Figure 9 - Data layer.

The Azure MySQL database runs on an Azure MySQL flexible server, that offers unlimited virtual storage and adjustable scalability to keep up with the business growth in terms of volume and number of transactions. Azure MySQL Flexible also offers options for automatic database replicas (Microsoft, 2025) done asynchronously for a read instancy, allowing for the use of a replica for reporting and analytics, without impacting the daily transactional component.

This polyglot persistence model balances strict transactional requirements with flexibility and performance. In the current implementation, ePharma adopts a single shared database instance across all services, rather than the more conventional approach of assigning one database per microservice. While this simplifies management, it introduces tighter coupling between services and a single database might not satisfy the data storage and access requirements for all services (Richardson, 2025).

1.10.4 Observability and Monitoring

Monitoring is achieved using Azure-native tools:

- Azure monitor to collect infrastructure metrics.
- Application Insights to provide distributed tracing and performance diagnostics.

- Log analytics to retain structures logs for investigation and auditing.
- Alert rules to proactively notify operators of anomalies.

1.10.5 DevOps and CI/CD

Development and deployment pipelines are fully automated using Azure DevOps:

- Boards and repos: Manage project planning and source control.
- Pipelines: Automate builds, tests and deployments.
- Artifacts: Store versioned dependencies and compiled components.
- Test plans: Execute automated functional and regression suites.

1.11Current Campaign Configuration Process

The Orders and Store functionality are the system's most critical features, that includes the campaign configuration process, that defines the business rules for product discounts, bonus quantities, and other promotional conditions that must be applied to customer orders. Revenue, customer satisfaction, and business performance are all directly impacted by this system component. Inaccurate pricing, missed promotions, or financial inconsistencies can arise from even a minor mistake in the campaign's setup, causing the client to suffer financial and reputational losses. This functionality requires a high degree of validation, precision and control over both the creation and edition of the campaigns.

Additionally, the ePharma system not only handles campaign configuration but also applies these configurations within the platform's order and store functionalities. Orders are directly dependent on the campaigns — meaning that both the configuration of campaigns and their application to products and order details are crucial for the platform's operation. This tight coupling between campaign logic and order processing further complicates the system's maintenance and adaptability.

Currently, the campaign configuration process in the ePharma system involves modifying the application's core business logic manually. When a client requests a new campaign or an update to an existing one, developers must manually implement those changes directly in the source code to reflect them in the actual campaign. Upon these modifications, the system is subjected to limited testing, often insufficient to guarantee correct operation. Once the testing phase is completed, the updated version is deployed to production.

The logic for both the configuration of campaigns and the application of their rules is implemented and retained entirely within the Order microservice. This service is responsible for interpreting the campaign data stored in MongoDB and executing the corresponding business rules during order processing. All rule definitions, conditional checks and enforcement mechanisms are embedded directly into the Order service's codebase, meaning that campaign behaviour is inseparably tied to this microservice. As a result, the Order service acts as the single

point of truth for campaign logic, centralising all decision-making and execution but also creating a strong coupling between campaign management and order handling.

Campaigns are implemented by developers as a mixture of procedural logic embedded in the application core and parameter records stored in the MongoDB database. Simple promotional parameters are stored in the campaign document but the behavioural semantics, the condition combinations, the eligibility calculations and the sequence of rule application are frequently implemented as explicit Java code. In other words, the system distinguishes between parameters (data stored in e database) and behaviour (code stored in the source tree). Because the deployment of behavioural changes requires code change and full release cycle, the system is effectively operated by developers rather than business users.

In practice, campaign configuration is split across two instances: descriptive data held in MongoDB (for example, minimum/maximum quantities, lists of mandatory products, eligibility flags and other parameters) and the behavioural code that intercepts and applies those parameters at runtime.

When the platform must apply the campaign logic to a certain order, the runtime process loads the campaign document and then executes the campaign's rules as implemented in the source code. These rule implementations are static: they are compiled Java code paths that contain fixed algorithms and conditional branches. There is no separate, first-class rule repository and no representation of an executable rule graph or decision table that could be independently versioned, tested and deployed. Consequently, any modification to the rules requires editing the source code, recompiling and performing a new deployment of the application.

Two characteristics of the current execution model are especially problematic. First, rule execution has no well-defined ordering semantics: the individual rule implementations are invoked without a formalised sequence or operator, and there is no explicit orchestration layer that guarantees determinism in rule application. In a domain where discounts and bonus may interact (for example, a percentage discount applied before or after a bonus calculation), the absence of a deterministic execution order can produce inconsistent or non-intuitive results. Second, the rules are extremely static because the logic lives in the code, personalisation is effectively infeasible or, at best, difficult and error prone. Introducing client-specific variations typically requires conditional branches in the codebase or evaluator classes, which increases complexity and risk.

To illustrate the absence of a well-defined execution order, the following code fragment shows how campaign rules are applied sequentially, without an orchestration mechanism that guarantees prioritisation among discount, bonus, and quantity checks.

```
void applyCampaignToOrder(Order order, Campaign campaign) {
   // Rules are executed sequentially in code
   applyDiscount(order, campaign);
   applyBonus(order, campaign);
   applyMinQuantity(order, campaign);
   applyMaxQuantity(order, campaign);
}
```

Code 1 - Execution order code example.

This arrangement produces tight coupling between campaign semantics and the order-processing pipeline. Campaign behaviour is directly connected with the code; as a result, a change intended for a single client frequently affects shared code paths and therefore risks unintended side effects for other clients. The lack of isolation and modularity means that a single change can cause system-wide repercussions.

The lack of protective automation, like tests or sandboxed user-facing environments that allow business users to simulate the effect of a change safely prior to publication is another characteristic of the campaign configuration process.

Operationally, the developer-centric change process compounds these architectural fragilities. Because changes to campaign logic are made by developers, frequently with limited domain expertise, the implementation of business requirements is mediated by interpretation. This mediation increases the probability that the delivered behaviour does not precisely match stakeholder intent. Moreover, because configuration parameters and code changes are not captured as immutable, versioned artefacts, there is no reliable historical record to reconstruct the exact rules set that applied to a past order. This absence of versioning and audit history makes reversion difficult and analysis labour-intensive.

For example, to alter the minimum required quantity for a specific product, an engineer must edit the parameter values stored in the campaign document and, depending on the complexity of the campaign, may also have to modify significant portions of the application code to accommodate special cases. Because campaign documents aggregate many per-product and per-entity configurations, they become large as the number of products and rules grows. Large MongoDB documents are harder to manage; they may degrade performance when documents contain many nested structures and repeated configuration blocks (MongoDB, 2025).

The following MongoDB document illustrates a simplified campaign. Although this is a reduced example, it provides insight into the structure and complexity of these configurations. In practice, campaign documents are significantly larger, often containing thousands of campaign entities and hundreds of product-specific configurations. As the number of entities and products increases, these documents become disproportionately extensive, making campaigns difficult to maintain.

```
{
  "_id": {
    "$oid": "68d6569196827e8d1d861d6b"
  },
  "campaignName": "Example Campaign",
  "clientId": "3037",
  "campaignStartDate": "2026-03-01",
  "campaignEndDate": "2026-09-30",
  "orderStartDate": "2026-01-01",
  "orderEndDate": "2026-12-31",
  "minNetPva": {
    "$numberDecimal": "100"
  },
  "maxNetPva": {
    "$numberDecimal": "2000"
  },
  "minQuantity": 50,
  "campaignProductIds": [
    "345341",
```

```
    "567567",
    "345345",
    "678678",
    "567844",
    "467568"
  ],
  "deliveredBy": [
    "56",
    "34",
    "23",
    "2"
  ],
  "campaignEntities": [
    "3674",
    "4334",
    "3475",
    "23",
    "3482",
    "4336"
  ],
  "discounts": [
    {
        "id": "345345",
        "minQuantity": 10,
        "maxQuantity": 25,
        "minDiscount": 10,
        "maxDiscount": 10
    },
    {
        "id": "678678",
        "minQuantity": 20,
        "maxQuantity": 50,
        "discount": 20,
        "maxDiscount": 10
    }
  ]
}
```

Code 2 - Campaign document example.

In addition to these structural and operational weaknesses, the system also suffers from adaptability and usability limitations. The current architecture was not designed to support the growing number of clients and frequent campaign updates required in the pharmaceutical industry. The reliance on developers for every configuration change creates bottlenecks that hinder growth and make it difficult to respond quickly to market demands. Furthermore, the absence of a user-friendly configuration mechanism prevents business users and marketing teams from managing campaigns independently, leading to inefficiencies and delays.

Therefore, this approach presents several challenges:

1. Developer-centric and time-consuming: The configuration process is highly time-consuming, since it requires the intervention of a developer, regardless of its complexity.

2. Error prone: Internal procedures for implementing changes are heavily dependent on human interpretation, increasing the risk of errors and reducing overall efficacity. This is highly error-prone, since manual modifications to business logic increase the likelihood of

inconsistencies. This can lead to unexpected system behaviour or incorrect discounts, bonus and other configuration-dependent offers.

3. Adaptability: The inflexible structure of the system makes it difficult to tailor campaign configurations to the specific needs of individual clients. As a result of this, applying custom business rules for different clients is cumbersome and requires extensive development work. With an increasing number of clients and growing demands for custom and very specific campaigns, the manual configuration is becoming unsustainable. The system's reliance on developer intervention creates bottlenecks that slow down response times and limit the company's ability to adapt to the evolving pharmaceutical market.

4. Static rule execution: Rule implementations are static, without a formalized execution order, which makes personalization difficult and can lead to inconsistent results when multiple rules interact.

5. Lack of isolation and versioning: Campaign behaviour is tightly coupled to the order-processing pipeline, with no independent rule repository or version control. As a result, changes cannot be safely rolled back, and historical configurations cannot be reconstructed.

6. Data management complexity: Campaign documents in MongoDB grow excessively large as more products and rules are added, which makes them difficult to manage and may degrade system performance.

These limitations, make the process neither adaptable nor sustainable. This not only increases the costs of development but also undermines the response time to market demands and conveys the users a lack of future-readiness and technological adaptability. Solving these limitations are the objectives of this thesis.

1.12Domain Analysis

The ePharma system originally lacked a formally defined or explicitly documented domain model. However, such understanding and systematization is of extreme importance to achieve the defined goals.

This section presents a comprehensive and structured model of the implicit knowledge embedded in the system's processes and logic. This endeavour required a deep analysis of the existing code, stakeholder interviews and the reverse engineering of business rules from various parts of the application.

1.12.1 Domain concepts

The domain of the ePharma Orders functionality is centred around the definition, validation and application of campaign and product specification to a given order. A clear conceptualisation of these entities and their relationships is essential to formalising a domain model that can serve as the foundation for the proposed solution.

At a high level, these abstractions can be identified:

- Order
- Order delivery
- Order Delivery Line
- Campaign
- Product
- Requester
- Supplier

The class diagram in Figure 10 illustrates these concepts, highlighting the main entities, some of their attributes and relationships. This diagram provides a structured visualization of the implicit domain model that supports the Orders functionality, offering a foundation for the design of the solution.

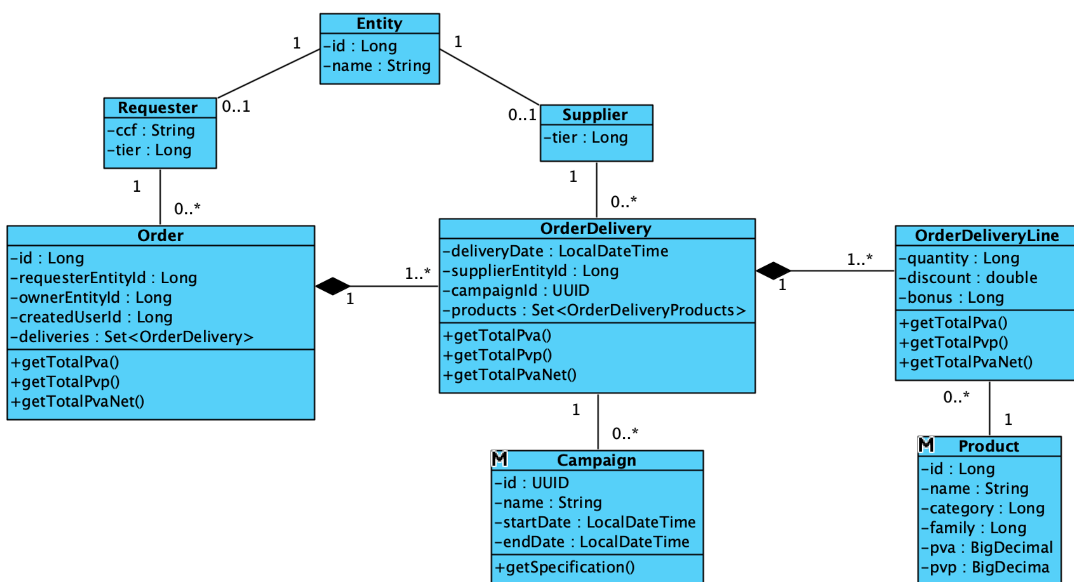

Figure 10 - Orders functionality class diagram.

1.12.2 Campaign specification

Campaign specifications are defined by a set of configurable dimensions that determine their scope, eligibility conditions and the way promotional rules are applied. These dimensions can be combined using logical operators such as AND, OR, XOR and NOT, allowing complex business rules to be modelled from relatively simple building blocks. Each dimension corresponds to a constraint or parameter that the system evaluates when determining whether a campaign is applicable to a given order or if it is correctly filled. Together, these dimensions constitute the functional foundation of campaign logic, capturing campaigns in a structured and machine-executable form. Typical campaign configuration dimensions include:

- Start Date: defines the date that the campaign starts to be active/available.
- End Date: defines the date that the campaign is no longer active/available.
- Delivery Start Date: defines the start date that the campaign can have delivery dates.
- Delivery End Date: defines the end date that the campaign can have delivery dates.
- Minimum Quantity: the minimum total product unit's quantity the campaign needs to be valid.
- Maximum quantity: the maximum total product unit's quantity the campaign needs to be valid.
- Minimum PVA value: the minimum PVA value the campaign needs to have to be valid.
- Maximum PVA value: the maximum PVA value the campaign can have to be valid.

The remaining dimensions are presented in Appendix A.

1.12.3 Campaign's Product Specification

Products of a campaign in the ePharma platform are described by a set of dimensions that capture their commercial, regulatory and logistical attributes. These dimensions form the foundation for campaign definition and order processing, since they determine how individual products can participate in promotions, pricing rules and eligibility checks. Constrains may apply to a single product or to groups of products in aggregate. For example, a rule may specify that a given product in a campaign must be ordered with a minimum quantity of 10 units, while another may state that the combined quantity of two or more products must reach at least 10 units.

This capability allows campaigns to capture both fine-grained product specific rules and broader category or bundle level restrictions, thereby increasing their flexibility and expressiveness. The product model must therefore be sufficiently rich to express both the intrinsic characteristics of the product and the extrinsic attributes relevant to business processes.

- Family: the family of the product of the campaign.
- Category: the category of the product of the campaign.
- Discount ranges by quantity: the discount ranges by quantity of the product in the campaign.
- Bonus ranges by quantity: the bonus ranges by quantity of the product in the campaign.
- Minimum Quantity: the minimum unit's quantity the product needs to be valid in the campaign.
- Maximum quantity: the maximum unit's quantity the product can have to be valid in the campaign.

- Minimum PVA value: the minimum PVA value the product needs to have to be valid in the campaign.

- Maximum PVA value: the maximum PVA value the product can have to be valid in the campaign.

- Mandatory: defines if a product is mandatory in the campaign.

The remaining dimensions are presented in Appendix B.

1.12.4 Systematization

The identification of campaign and product in campaign dimensions allows for their consolidation into a structured and concise systematization. This step is of extreme importance for the transformation of the extensive list of attributes into a coherent model that can be used as the foundation for the DSL vocabulary and for the rule engine's configuration logic. The systematization highlights the essential parameters that govern campaigns and its products in the ePharma system, grouping them into logical categories that emphasise their role in defining eligibility, scope and constraints.

1.12.4.1 Campaign specification

For campaigns the systematization captures client association, temporal boundaries, quantities and financial constraints, as well as override and validation requirements. These elements together determine whether a campaign can be applied and under what conditions.

- Client ID

- Start Date [Active | Delivery]

- End Date [Active | Delivery]

- Minimum [PVA | PVA NET | PVP | PVF | Discount | Bonus | Quantity | Product Families | Product Categories| Deliveries]

- Maximum [PVA | PVA NET | PVP | PVF | Discount | Bonus | Quantity | Family | Category | Deliveries]

- Manual Override [Discount | Bonus]

- Mandatory Superior Validation

1.12.4.2 Campaign's Product Specification

For products, the systematization consolidates the commercial and regulatory parameters that shape how individual or grouped items participate in promotions in a given campaign. Discount and bonus ranges, multipliers and minimum/maximum thresholds are central to ensuring the validity of product-level rules and to enabling the combination of products into flexible campaign configurations.

- Family
- Category
- Discount ranges [Quantity | PVA | PVP | Tier]
- Bonus ranges [Quantity | PVA | PVP | Tier]
- Bonus multiplier [Simple | Range | Tier]
- Minimum [Quantity | Quantity by month | Quantity by delivery | PVA | PVP | PVF | PVA NET
- Maximum [Quantity | Quantity by month | Quantity by delivery | PVA | PVP | PVF | PVA NET]
- Mandatory

1.12.5 Core Domain Entities

Based on section 1.12, this section presents a UML diagram that formally captures the previous information and knowledge.

It is important to emphasize that what follows is not a pre-existing model merely transcribed, but a structured synthesis developed through careful domain exploration and abstraction.

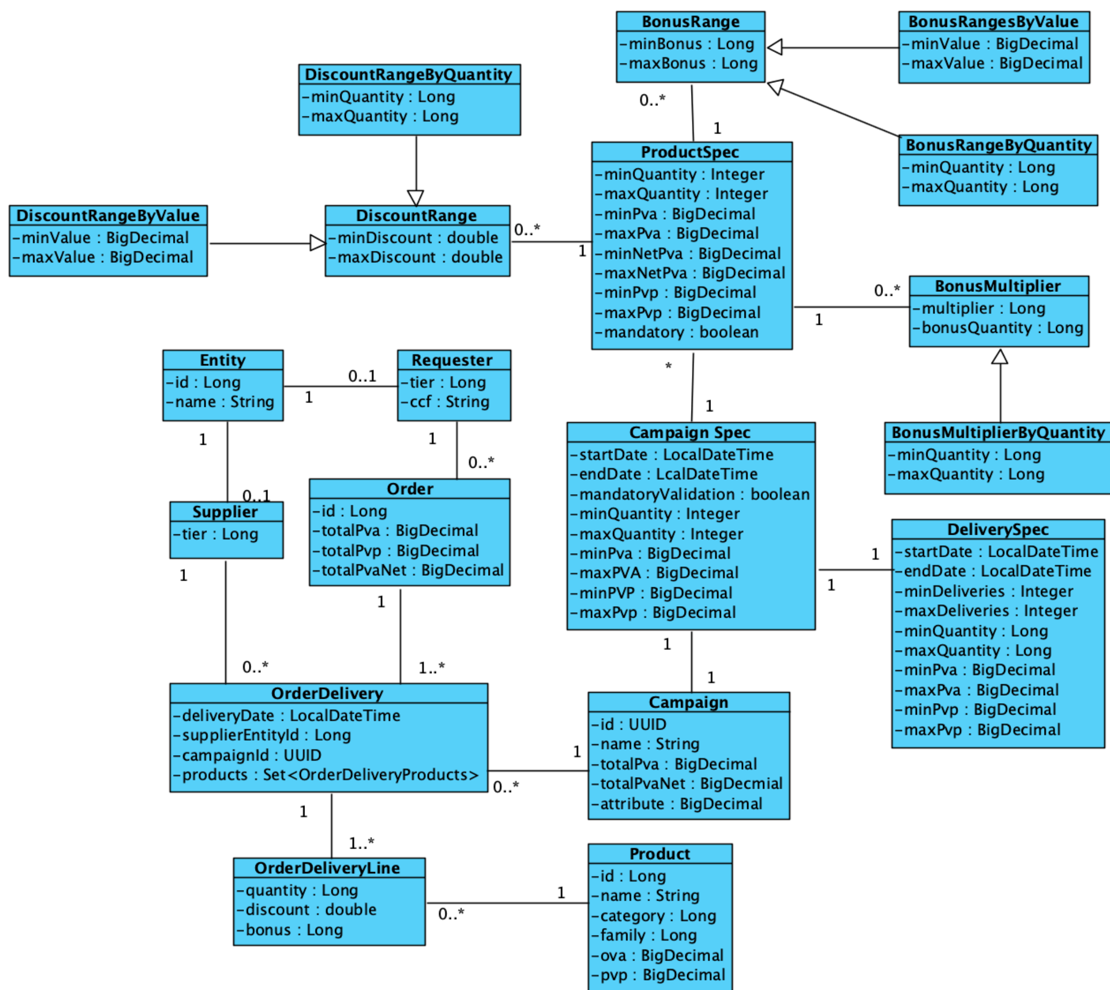

Figure 11 – Analysis domain diagram.

At the centre of the domain lies the Campaign, that represents a commercial initiative that dictates bonus, discount and many other conditions over a defined time frame that must be met to ensure its usage. A campaign is composed of various sub-specifications that define business rules across various axes, and it represents a commercial strategy. Each campaign has a given Campaign Spec, with the constraints that shape how and when a campaign is activated or available, and under which conditions it applies.

Campaigns specs are associated to multiple campaign's products specs, requesters specs (the entity requesting the products, e.g. pharmacies) and delivery rules.

Products are central to campaign logic, each is identified by a code and contains multiple price dimensions, such as PVA, Net PVA, PVP, and more. Similarly to the campaigns, the products have a campaign's Product Spec, which define the campaign specific boundaries for each product. These may include limits on either quantity or value, and it also supports complex discount and bonus relationships that define specific bonus or discount thresholds based on an arbitrary field, like quantity or value (e.g. PVA, PVP), enabling the configuration of highly customizable promotional strategies.

Bonus logic can be defined in several ways; it can use ranges that specify minimum and maximum bonus quantities based on a quantity or value range. It can also be configurable using multipliers, where for each 10 units, 2 bonus units are available, using multiplier parameters to calculate bonus dynamically.

Discount logic can also be applied in ranges, similar to bonus logic. These allow configuration of rules that can give different discounts based on quantity or value ranges. These are also linked to a given product spec, providing flexible and granular control over discount application.

An order is the top-level entity; it can apply multiple campaigns and have multiple delivery dates. Delivery dates specify the date of the delivery; the user can configure multiple delivery dates and distribute the quantities as s/he pleases. Deliveries are evaluated through Delivery Spec, which contains date ranges, value thresholds, and quantity limits. This spec becomes extremely valuable when delivery-based validations is enabled at the campaign level.

The requester represents the entity that places the order, usually a pharmacy, reseller or distributor. The supplier is associated with a given OrderDelivery, representing the supplier that will deliver that given delivery date.

1.13Constraints

Functional constraints must carefully align with the goals of extensibility, maintainability and business agility. These constraints focus not only on the capabilities that the system must provide, but also on how these capabilities can support broader architectural principles like modularity, separation of concerns and user empowerment. The functional design will be crafted to accommodate the dynamic business environment that the pharmaceutical industry presents, where marketing strategies, regulatory constraints, and client-specific campaigns can evolve rapidly and independently of the software development lifecycle.

Normally, constraints are defined following a standard classification, like FURPS+ and ISO/IEC 25010. Both classifications are mature and well proven, however the categories defined under FURPS+ for Information Systems by Robert Grady at Hewlett-Packard will be considered.

FURPS+ is an acronym for Functional, Usability, Reliability, Performance, Supportability and the extension (+) emphasizes other subfactors, such as Implementation and Interface (Larman, 2004).

Based on the goals and requirements presented in previous chapter and following the categorization of FURPS+, the identified constraints are:

- Functionality defines functional requirements, considering requirements that are not represented by specific use cases and apply to the entire system:
    - Dynamic Rule Definition: The system must allow users to define business rules dynamically though a domain-specific language expressed in JSON. The DSL should serve as a declarative abstraction upon rule logic, enabling the definition of both atomic and composite rules. The structure must be both intuitive and predictable, allowing stakeholders to configure campaigns without the need of deep technical expertise.

- Rule Composition and Logical Operators: The DSL must support composability through logical operators, such as AND and OR, but not limited to these, as more operators can be used. These compositions must be nestable to an arbitrary depth and supporting recursive evaluation semantics. This ensures that users can express multi-condition validation rules within a single coherent structure.

- Runtime Rule Evaluation: Rules must be evaluated in real time, during application execution, using the current operational context. This context may include information such as product inventory, user information, order details and the campaign metadata.

- Domain-Agnostic Support: The DSL and rule engine should be sufficiently abstract to support various domains beyond product rules. These can include user segmentation, delivery constraints, thresholds or temporal conditions. This generalization ensures that the designed solution remains future-proof and adaptable to new use cases.

- Usability defines the requirements that ensure that the product is understandable, easy to learn and use:

    - The Rule implementations must be easily understandable, editable and traceable.

- Reliability defines the ability that the product has to perform a given function/action under specified conditions:

    - Rule definitions are externally configurable, being dependent on user input or third-party integrations, therefore input validation and schema enforcement are essential for risk mitigation, such as injection attacks or data leaks. Error handling must include a comprehensive set of failure messages, auditing logs and fallback mechanisms to ensure operational transparency and resilience.

- Performance specifies to which degree the system performs its designed functions.

    - The system must be able to scale effectively to accommodate the increasing number of clients, products and campaigns. This scalability includes both horizontal and vertical scaling, that include deploying rule evaluation components across multiple instances and the optimization of in-memory evaluation to handle larger rule trees.

- Supportability defines the ability of the system to be operated during its life cycle, like adaptability, configurability and maintainability:

    - Extensibility and modularity: The rule engine must follow a pluggable architecture, where new rule types can be introduced seamlessly. This includes the support for dynamic discovery of rule classes, ensuring that the core engine remains untouched while facilitating the independent development of rule modules. Each rule must function as a self-contained unit, that can be composed, reused, or replaced without affecting the overall system integrity.

- The engine must integrate with the existing microservices application in the ePharma architecture, leveraging common data models and service APIs.

- Separation of Concerns: The business rules must be defined and maintained in complete isolation from the domain model. This promotes clear architectural boundaries, significantly reduces the risk of unintended side effects and simplifies reasoning about the system behaviour. The DSL must serve as a vessel, a declarative layer that interfaces cleanly with the domain context, without embedding business logic into the core codebase.

By fulfilling these constraints, the system will not only meet the needs of campaign configuration but also ensures a long-term reliability and user satisfaction within the ePharma system.

Design

The design of the proposed solution builds directly on the analysis of the ePharma domain, and the functional and non-functional constraints identified in the previous chapters. The primary objective is to provide a robust, extensible and maintainable architecture that overcomes the rigidity of the current campaign configuration system.

This section introduces the conceptual and technical blueprint of the solution, detailing how a Domain-Specific Language and a supporting rule engine are employed to decouple business logic from application code while enabling stakeholders to define, validate and manage campaigns in a structured and business user-friendly way. By systematically presenting the approach, constraints, core principles, architectural components and language specification, this section establishes a comprehensive foundation that both justifies the architectural choices and enables replication or extension in future implementations.

1.14Architecture

As no design approach was specified in advance by requirements or constraints, a Domain-Specific Language (DSL) quickly emerged as the dominant choice, as it provides a structured and intuitive way of defining and applying campaign logic, which conceptually reduces errors and streamlines the authoring process. This would improve operational efficiency, increase system flexibility, and ultimately enhance the overall adaptability and supportability of ePharma's platform while reducing long-term costs.

This approach implements the design principles introduced in Section 1.4; it separates rule data from rule behaviour, exposes campaign configuration through a high-level declarative DSL, and executes campaign logic in a dedicated, versioned and observable way.

Figure 12 illustrates the activity diagram of campaign authoring, from the moment a user defines or edits a campaign to the point at which it is executed. The paragraphs that follow justify the architecture and then describe each element of the flow in technical detail.

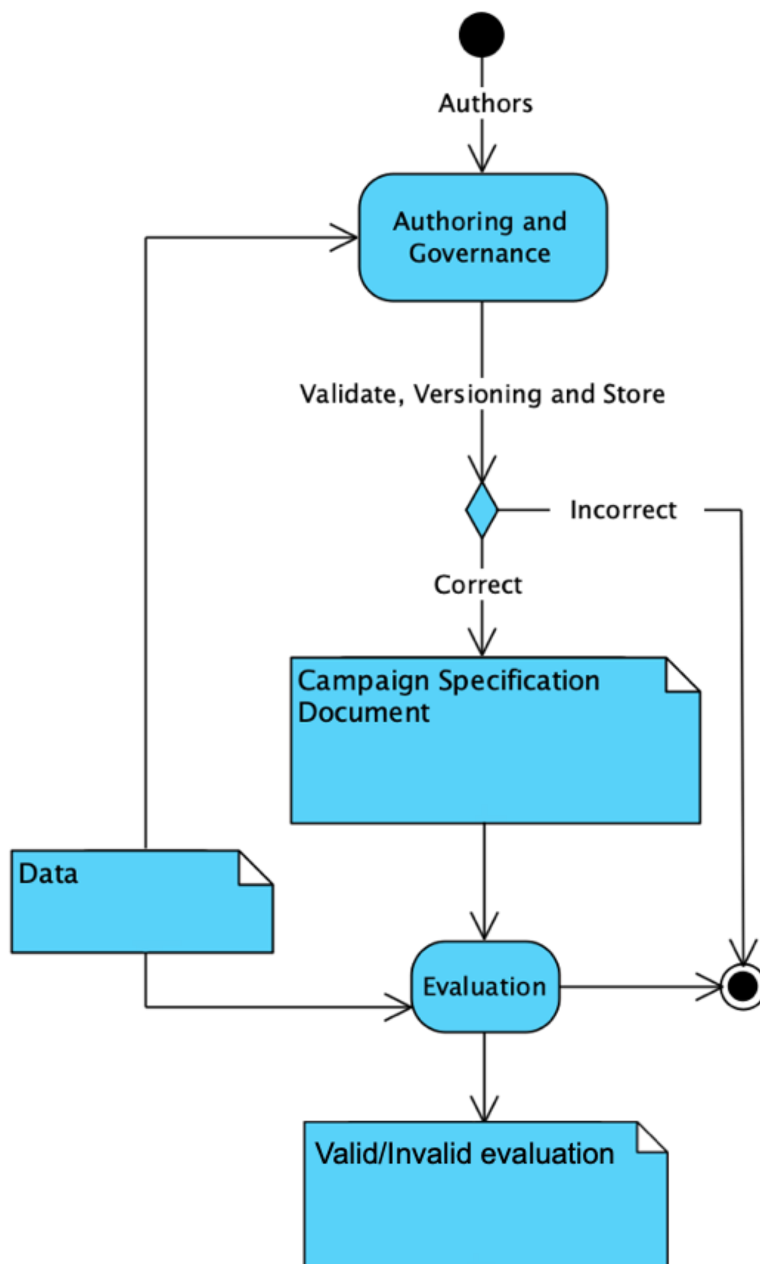

Figure 12 - Activity diagram of campaign authoring and evaluation.

The process begins with the business user or analyst, who authors a campaign through a dedicated authoring interface, this can take the form of a graphical UI or the direct editing of a JSON specification. Campaigns are expressed in the DSL, which captures the business constraints and campaign logic in a machina-readable form, but still in a human-readable format.

Once authored, a campaign undergoes a series of validation steps within the Authoring and Governance module. The first layer is the JSON Schema Validator, which ensures that the campaign conforms structurally to the DSL specification. This validation enforces the

correctness of fields, types and nesting, thereby preventing malformed inputs from propagating further into the process. This provides a mechanism for detecting logical inconsistencies, conflicts or unintended consequences before deployment.

A key aspect of this phase is the Validation component, which operates both at design-time and at runtime. During design-time, it provides a set of structural and contextual checks to ensure campaigns are syntactically correct and semantically meaningful. This includes validating references to products, ensuring correct thresholds are within acceptable ranges and verifying the consistency of composite rules, such as AND/OR combinations. The component also supports behavioural validation via simulation, which allows rules to be tested against sample or historical data. During runtime, the validation ensures execution consistency by checking type safety and rule semantics when campaigns are loaded and executed by the engine. Together, these validation steps establish a robust safeguard that increases both reliability and trust in the DSL. This directly mitigates the lack of automation in the legacy system, while introducing an execution order and logic, thereby avoiding the inconsistent outcomes previously observed.

After validation, the Versioning and Change Control module manages the lifecycle of campaigns. Each modification creates a new version, enabling traceability and rollback if necessary. Once a campaign version is approved, it is published through a Campaign Publisher, which serves as a store and rule registry. This ensures that a consistent, versioned campaign specification is available for consumption at runtime, and that rollbacks can be done if needed. This resolves the former lack of versioning and audit history, guaranteeing the campaigns can be traced and reverted at any time.

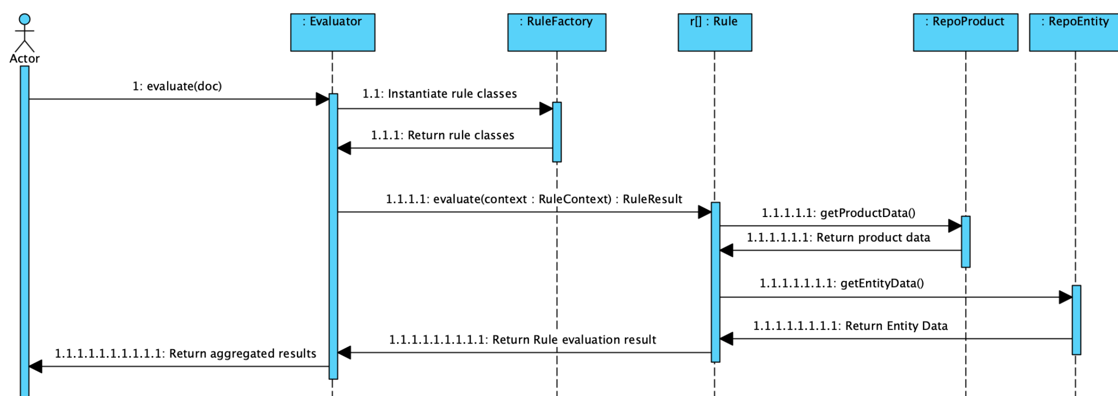

Figure 13 - Evaluation sequence diagram.

At runtime, the Rule Engine Service retrieves the campaign specification from the registry and delegates it to the Evaluator, as shown in Figure 13. The Evaluator is responsible for interpreting the DSL constructs, instantiating the relevant rules classes via the Rule Factory, and assembling them into an executable rule set. To evaluate rules in the correct domain context, the needed data is fetched from external services – such as Entity Service, Product Service, Order Service and User Service – and constructs a runtime representation of the evaluation context. This modular runtime evaluation replaces the previous static rules, allowing campaigns to be personalised per client without embedding conditional branches into the application code.

The evaluator then executes each rule against the context, producing results along with explanatory reasons for pass or fail outcomes. These results are returned to the actor, which

uses them to enforce business constraints, determine campaign applicability, or reject invalid orders. By isolating campaign logic from the transactional order-processing pipeline, this design eliminates the tight coupling of the existent system and ensures that campaign behaviour can evolve independently of the core application.

Through this combination of authoring, validation, governance, publication and runtime evaluation, the implementation ensures that business rules remain consistent, auditable and enforceable across the entire order management process. The integration of a validation pipeline provides a robust foundation for deploying complex business policies in a mission-critical environment.

1.15Core Design Principles

The architecture and the design of the DSL and the corresponding rule engine are guided by a given set of foundational principles, aiming to ensure flexibility, extensibility and maintainability. These principles are a crucial part of the system, to support the dynamic nature of campaign configuration in the ePharma platform, which must rapidly adapt to the diverse client requirements and frequent changes in the business.

The core of the design is the philosophy of treating each rule as a discrete, self-contained module, completely independent from the system's core domain logic. The modularity that this approach presents, enables rules to be added, removed or updated dynamically and per client – without requiring modifications to the rule engine's internal mechanics or to the domain model. This plug-and-play architecture not only provides flexibility to construct complex rule sets in a composable manner but also allows for rapid reconfiguration of business logic without the risk of introducing instability into core components.

The rule evaluation logic is entirely decoupled from domain entities, persistence mechanisms and procedural business logic. Instead, rules are defined and managed through a declarative DSL component distinct from the core platform. This architectural separation ensures that the logic captured in the rules can evolve independently of the rest of the system, allowing domain experts and business analysts to create, understand, and manage rules without direct developer involvement. This reduces the cognitive load on developers, facilitates faster iteration cycles and minimizes the risks of regressions in unrelated parts of the system.

The system leverages a declarative registration mechanism in conjunction with runtime introspection to automatically discover and register rules. By using this pattern, the need for manual wiring of new rules into factories or configuration files is eliminated. Developers can simply define a new rule class, annotate or declare it appropriately, and the rule engine will register it for further use in campaign specifications. This approach allows the configuration of campaigns to grow organically as business needs evolve, thereby streamlining the development process.

To ensure a fully customizable solution, the rule engine supports composability via logical operators, such as AND, OR, and potentially XOR, enabling the creation of hierarchical and tree-like structures. These operators consist of rules that can recursively contain other rules, allowing arbitrary deep and complex logical expressions, supporting a rich and expressive rule vocabulary, while still maintaining the simplicity of individual rule definitions.

Furthermore, to evaluate consistently and in isolation, the system introduces a context object that encapsulates all relevant data inputs (e.g., product details, client metadata, campaign parameters) required for a given evaluation. This context abstraction, ensures that rules are stateless, promoting cleaner design, easier testing, and greater reusability across different use cases. Also, by centralizing data access though the context, the system also gains transparency and auditability in rule execution. Robust error handling and validation is also a core principle in this design, where each rule is responsible for validating its own parameters during instantiation. This includes checks for required fields, data types, and business constraints. In the event of invalid input or failure during evaluation, the rule engine returns comprehensive error messages. A robust error reporting framework is imperative to ensure operational transparency, facilitate debugging, and that misconfigurations are identified and addressed promptly before affecting users.

1.16DSL

The structure of the rule engine is based on several well-defined components, each with a specific responsibility. This modular design promotes separation of concerns, extensibility and maintainability (Hudge, 2024). It enables the dynamic evaluation of business rules, defined through a Domain-Specific Language (DSL), allowing a flexible configuration of campaigns, in a decoupled approach.

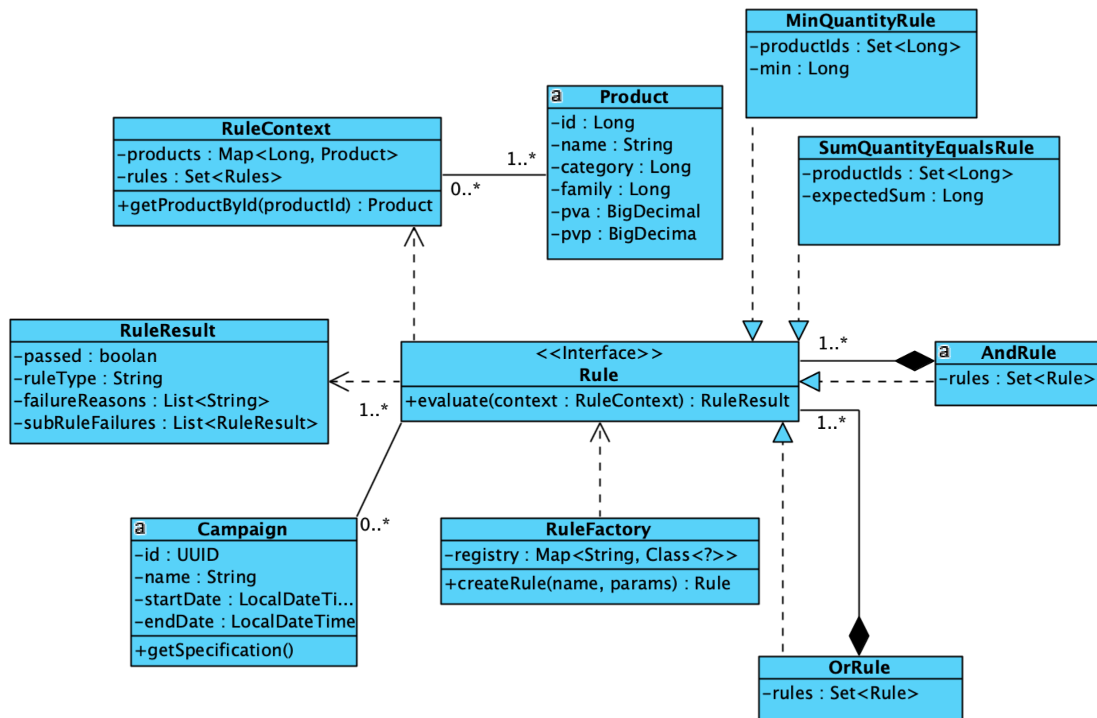

Figure 14 – DSL class diagram.

As shown in Figure 14, all rules must adhere to a common interface - the Rule interface – which declares the evaluate(RuleContext) method and returns a RuleResult object. This interface is

the base for all rules. They must implement this interface to ensure consistency across different rule types and still promoting polymorphism and testability.

A further advantage of this architecture is its extensibility through abstraction. The rule engine itself operates exclusively against the Rule interface, remaining agnostic to the actual implementations of specific rules. The dependency arrows in the class diagram emphasize this design: concreate classes such as MinQuantityRule implements the Rule interface. This means that the core engine does not need to be modified or recompiled when new business requirements emerge. Instead, developers can introduce new rule types simply by creating additional classes that implement the Rule interface. Once registered, these new implementations can be immediately consumed by the engine in the same way as existing rules. This design embodies the Open/Closed Principle, ensuring that the system is open to extension (e.g. new rules) but closed to modification (e.g. the rule engine only considers the Rule's implementations). From an engineering perspective, this promotes a sustainable evolution of the platform: the rule engine maintains stability and testability, while the rule vocabulary can expand organically to cover new business cases. For a domain as dynamic as pharmaceutical campaigns, this decoupling of interface and implementation is fundamental.

The implementations of the Rule interface, the concrete rules, encapsulate the validation logic for specific business constraints. For instance, a MinQuantityRule may enforce that a specific product must have a minimum order quantity. These rules are designed to operate in isolation, drawing necessary information from the RuleContext.

Composite rules are a cornerstone of the rule engine's composability principle. These rules allow the definition of logic using operators like AND and OR. Each composite rule maintains a collection of subrules and evaluates them recursively based on the logic it implements. This recursive strategy enables rule nesting, allowing the construction of complex business logic to be expressed cleanly and hierarchically. For example, a composite rule might require a combination of two or more different rules, each represented by subrules that are independently evaluated but logically connected.

The RuleContext serves as the abstraction layer between the data sources and the rule engine. It encapsulates all relevant runtime data, such as product specifications, user details, delivery details, and other information. By providing a centralized access point for all evaluation-related data, the RuleContext ensures that the individual rules remain stateless, which greatly enhances testability, concurrency safety measures and separation of concerns. Moreover, it exposes convenient query methods, like fetching product data by ID, further simplifying rule logic.

All domain entities feed the RuleContext, a runtime object that encapsulates the current state of the system. This includes user, product and campaign data, but can contain various types of information needed for the correct execution of the rules. This structure enables the rule engine to dynamically evaluate the constraints, such as:

- Constraints based on purchase history, that rely on Requester Spec and historical order data.

- A bonus eligibility check, that may combine ranges and multipliers.

- Check certain limits using Product Spec, Campaign Spec, Order Spec or even Delivery Spec.

This modular design aligns naturally with the DSL and the rule's engine architecture, allowing for extensibility and isolation. Each rule can be composed, nested or modified independently, and the domain model supports the dynamic composition through its clean separation of concerns and contextual data encapsulation.

The RuleResult is an object that has a very important impact in the reporting and diagnostic tools for the rule engine. This object captures the outcome of a rule evaluation, indicating whether the rule evaluation succeeded, the executed rule, and messages detailing the failure causes. This structure facilitates inspection and transparency, which is critical in complex commercial scenarios. It allows users, developers and even external systems to understand exactly which conditions didn't pass, enabling precise debugging and intelligent feedback for the end users.

1.16.1 Syntax

To address these limitations of the current campaign configuration system – particularly in terms of flexibility, maintainability and adaptability – a Domain-Specific Language (DSL) is proposed as a fundamental architectural element. The main objective of the DSL is to abstract campaign logic from the application's source code into a dedicated, high-level language that allows for a more intuitive and decoupled expression of the configuration process. The goal is for this tool to be used not only by developers but also by business stakeholders such as marketing teams and product managers.

The DSL is designed to be accessible to both technical and non-technical stakeholders, by adopting familiar formats such as JSON, XML or YAML. This section defines the formal semantics of the DSL used for expressing campaign rules in the specification engine.

The abstract syntax defines the structural foundation of the DSL, describing the core elements and their relationships independently of concrete textual or graphical representations (Gomez-Vazquez and Cabot, 2025). In ePharma's campaign configuration system, the abstract syntax provides a formal and implementation-agnostic model to represent the campaigns and the rules that govern them.

A campaign is represented as an aggregate of rules, each encapsulating a business constraint to be enforced at runtime. These rules are expressed through a unified interface and are typed, meaning that each rule instance belongs to a specific rule class, such as quantity validations, eligibility filters or date constraints. Rules can be recursively composable through logical operators, enabling complex campaigns to be built from smaller and reusable components. This structure supports a declarative style of configuration, where business intent is modelled through the composition of a discrete set of rules statements.

The abstract syntax distinguishes Atomic Rules and Composite Rules. Atomic Rules defines indivisible conditions, such as minimum quantity or delivery date ranges. These operate directly over field extracted from the runtime context. Composite rules, on the other hand, define structural combinations of atomic or other composite rules using logical operators, producing nested evaluation trees that mirror the logical dependencies of the underlying campaign.

To support parameterization and domain specificity, each rule may receive named parameters, like threshold values, multipliers and date ranges, that define its operational context. These

parameters are not interpreted in the abstract syntax level but are structurally included to be resolved later during the evaluation of the rule.

Each rule is associated with a target domain concept, representing the entity or attribute it validates (e.g., a product's quantity, an order's total value or a requester's classification). This linkage ensures the rules are semantically aligned with the business domain and can be validated in isolation or within the scope of a campaign. The abstract syntax also supports rule scoping, where rules are contextualized to apply only under specific conditions or entity scopes, such as a particular platform, region or tier. This allows for a more selective application of rules and promotes modularity, enabling the reuse of rule definitions across campaigns, ensuring minimal duplication.

The abstract syntax is modelled programmatically as a class hierarchy rooted in a Rule interface, from all rule types inherit. This inheritance structure allows the engine to treat rules polymorphically, enabling uniform evaluation logic and a high degree of extensibility.

To better visualize the structure of this syntax, Figure 15 presents an Abstract Syntax Tree (AST) representation of a sample campaign configuration. This example depicts a campaign where various rules are applied: a minimum quantity rule and a composite OR rule, both combined though a logical AND. The OR rule implements the OR operator to two different rules, sum quantity and data range validation rules.

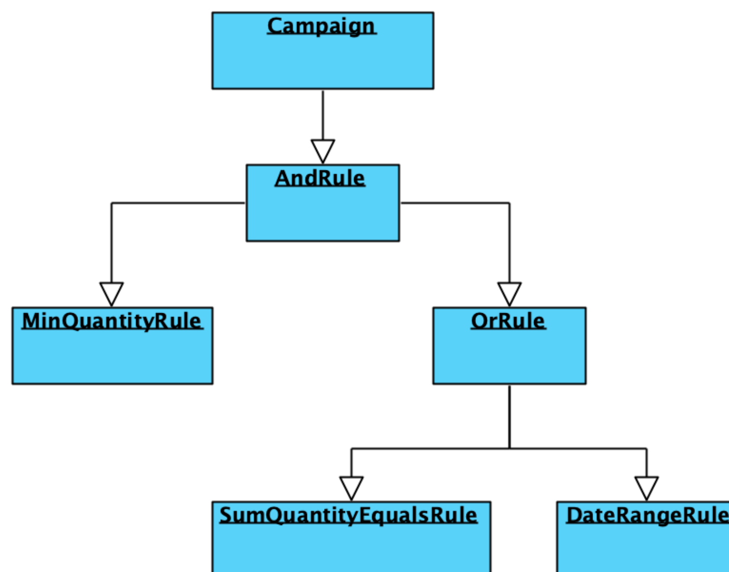

Figure 15 - Abstract Syntax Tree of a campaign example.

In summary, the abstract syntax of the DSL provides a declarative model for campaign configuration. It formalizes the conceptual constructs of the language – rules, compositions, parameters and scopes – independently of syntax details, ensuring that business logic is encoded in a clear, modular and semantically consistence manner. This foundational structure plays a central role in making the system extensible, maintainable and robust against future domain evolution.

1.16.2 Semantics

The semantics of the DSL provide a formal way of defining the meaning associated with its constructs. While the abstract and concrete syntax, capture the structural composition of the rules and the describe how they are expressed, respectively, the semantic layer defines how these constructs are interpreted and evaluated over the domain-specific context of the ePharma system. This interpretation is imperative for ensuring correctness, traceability and alignment with the business logic (Keshishzadeh and Mooij, 2014).

In the proposed DSL, semantics are, in nature, operational – that is, they specify how rules behave during execution, given a particular input context. Each rule evaluates against a RuleContext, a structured representation of relevant business entities, such as products, quantities, campaign specification and requester constraints. The evaluation process yields a Boolean outcome, typically represented as either a pass or a failure, accompanied by diagnostic metadata.

Atomic rules encapsulate basic business predicates, e.g. MinQuantityRule and SumQuantityEqualsRule. Formally, let Q(p) denote the quantity of product p in the order. Then the semantics of a MinQuantityRule over a set of products P and threshold t can be expressed as:

$$MinQuantityRule(P, t) \iff \forall \mathrm{p} \in \mathrm{P},\ \mathrm{Q(p)} \geq \mathrm{t} \quad (1)$$

Atomic rules thus serve as the semantic foundation of the DSL, mapping domain constraints to logical predicates.

Composite rules enable the expression of more complex logic, by combining atomic or other composite rules using logical operators. The DSL supports standard propositional connectives, including AND and OR, but other operators can be used.

Given a composite rule $\mathrm{R} = \mathrm{AND(R1, R2, \ldots, Rn)}$, the semantics are defined recursively:

$$Eval(R) = \bigwedge_{i=1}^{n} Eval(Ri) \quad (2)$$

This recursive interpretation guarantees composability and supports arbitrary nested logical expressions. The semantic model also includes short-circuit evaluation and error propagation mechanisms to optimize performance and improve interpretability.

All rule evaluations are stateless and deterministic – they do not rely on mutable internal state or external side effects. This ensures that rule outcomes are fully determined by the input RuleContext, promoting referential transparency and facilitating automated reasoning and testing.

Furthermore, the semantics engine produces evaluation traces, detailing which rules were evaluated, their results, and any intermediate failures. These traces serve both for diagnostic and audit purposes, allowing business stakeholders to understand why a rule passed or failed, and to validate that the behaviour of the rule set aligns with organizational policy.

The semantic design of the DSL is closely aligned with ePharma's domain requirements, particularly the need for correctness, clarity and adaptability in campaign configuration. By formalizing rules as composable predicts over domain entities, the DSL provides a reliable, stable and interpretable mechanism for enforcing the commercial logic defined by the stakeholders.

In summary, the rule semantics layer offers a rigorous and transparent foundation for decision-making within the ePharma platform. It bridges the gap between high-level business requirements and executable system logic, ensuring that campaigns are not only expressive but also easily verifiable, maintainable and semantically correct.

1.17Summary

The class model presented in previous diagram of the ePharma system is comprehensive and structured to support complex business rules, allowing for a high degree of configurability by supporting multiple levels of abstraction, ensuring that the rule engine can validate campaign logic accurately and dynamically. This strong domain foundation is essential for enabling adaptability, maintainability and agility.

It is essential to emphasize that the present examples and constraints do not represent the totality of the business logic present or supported within the system. In fact, the variety and volume of potential constraints in campaigns are too extensive and domain-specific to be fully catalogued.

The introduction of a DSL comes from the need to overcome the system's current rigidity. The DSL is deigned to be expressive enough to represent the complex business rules currently supported by the system, while remaining simple and accessible for non-technical users. It is constructed to support modular and reusable definitions, allowing rules to be composed using logical operators, and supports parameterized rule types that can be dynamically instantiated and evaluated. The language is extensible, meaning that as new requirements emerge – such as new campaign types, product specifications or delivery constraints – new rules can be added to the DSL vocabulary without requiring architectural changes – a foundational goal of this approach.

Another key benefit of the DSL-based approach is its impact on maintainability. Rules can be validated independently, allowing teams to conduct evaluations using sample data before deploying to production. This provides a high level of confidence in the accuracy of the configurations and significantly reduces the likelihood of errors reaching the production environment.

This redesign also allows for the creation of a rule authoring and management user interfaces, where users can define campaign logic through guided forms or visual builders that automatically generate the underlying (but independent from UI) JSON DSL code. This interface acts as a graphical abstraction over the DSL, allowing users to construct the campaigns through guided forms or drag-and-drop components, without requiring direct interaction with the underlying JSON syntax. This approach enables business users to draft, validate and approve campaign logic in an end-to-end workflow, reducing the time for campaign updates, promoting consistency and reducing the redundancy of the system.

It is important to emphasize that the principles and advantages described are a direct consequence of the functional and non-functional constraints outlined in Section 1.13. By making this relationship explicit, the adoption of the DSL can be understood not merely as a design choice, but as a deliberate response to the defined constraints, thereby ensuring coherence, traceability and justification for the proposed solution.

Implementation

The implementation phase translates the conceptual design of the DSL and its supporting rule engine into a concrete, operational system within the ePharma platform. This stage validates the theoretical assumptions made during the design phase and demonstrates their feasibility in practice. This section discusses the main findings observed during the development process, details the technical implementation, highlights the benefits, reflects on the limitations encountered and compares the solution with existing approaches.

1.18DSL concrete syntax

The DSL is declarative, data-driven, and represented in JSON. It is designed for human readability, composability, and ease of integration with client-side and back-office tools.

An implementation decision is to express the DSL in JSON syntax, providing a lightweight and widely adopted format that is both machine and human readable. This syntax provides a straightforward integration with web-based tools, simplifies validation through JSON schemas and supports interoperability with third-party systems. Furthermore, JSON's tree-like structure naturally accommodates the hierarchical and compositional nature of campaign rules, where atomic rules can be combined into complex logical expressions.

While alternative formats such as XML or YAML could have been adopted, JSON was chosen deliberately for this thesis. XML, although expressive, introduces considerable verbosity that reduces readability and increases the cognitive load for non-technical users. YAML, on the other hand, is more concise but prone to syntactic fragility due to its reliance on indentation, making it more error-prone. JSON is balanced by being widely supported across programming environments, simple for business stakeholders to read and easily validated using existing schema technologies. It should also be emphasized that the architecture of the rule engine does not restrict the use of JSON exclusively, the DSL could be serialised in other formats if required by integration or business needs. However, JSON was selected in this work for its clarity, ecosystem support and alignment with the usability and maintainability goals of the ePharma platform.

```
}
  "rule": {
    "type": "OR",
    "params": [
      {
        "type": "MinQuantityRule",
        "params": { "productIds": [1], "min": 5 }
      },
      {
        "type": "AND",
        "params": [
          {
            "type": "MinQuantityRule",
            "params": { "productIds": [1], "min": 2 }
          },
          {
```

```
                "type": "SumQuantityEqualsRule",
                "params": { "productIds": [1, 2], "sum": 10 }
              }
            ]
          }
        ]
      }
    }
```

Code 3 - DSL concrete syntax example.

This example demonstrates the composability of the DSL. The outermost OR operator specifies that the campaign condition will be satisfied if either of the following passes:

1. Product with ID 1 is ordered with a minimum quantity of five units.
2. Both product with ID 1 has at least two units and the combined quantity of products with IDSs 1 and 2 equals 10 units.

Such nested logical constructs illustrate the expressive power of the DSL while maintaining readability and predictability for both technical and non-technical stakeholders.

These constructs can be mapped to the dimensions specified in Section 1.12. The construct MinQuantityRule directly corresponds to the minimum quantity per product dimension, enforcing product-specific thresholds that govern campaign eligibility. The rule SumQuantityEqualsRule represents group-based constraints, also discussed in Section 0, where eligibility can depend on the aggregate quantity of multiple products rather than an individual item. The parameter productIds aligns with the product unique identifier dimension, ensuring a direct mapping between the DSL and the underlying product catalogue.

While the abstract syntax defines the language's concepts and their allowed combinations, concrete syntax defines how those concepts are presented to the user, either through a graphical or textual notation (Meyers et al., 2017). It determines how users express rules in a human-readable and machine-parsable format.

The concrete syntax allows non-technical users to define, visualize and modify commercial rules without the need to directly interact with the source code. The DSL's JSON-based structure mirrors the abstract syntax tree but introduces a practical and usable way of composing rules, validation logic and integrating these rules into the campaign configuration workflow.

Each rule definition specifies its type and its parameters. For composite rules, the parameters, are other rules, that are applied with a given operator. This approach allows for recursive composition of logic using fundamental building blocks, all expressed in a clear and consistent way.

For example, a simple atomic rule to ensure that the quantity of a specific product is not below a defined threshold might be expressed as:

```
{
  "rule": {
    "type": "MinQuantityRule",
    "params": { "productIds": [1], "min": 10 }
```

```
  }
}
```

Code 4 - Atomic rule concrete syntax example.

This structure encapsulates a constraint stating that at least 10 units of the product with the identifier 1, must be present in the given campaign. Multiple products can be passed; in this case all of the products must meet this condition.

Composite rules allow for the combination of multiple subrules using logical operators. For instance, to enforce that the minimum quantity for a given product is met and the total quantity of a group of products matches a specific value, the syntax would be:

```
{
  "rule": {
    "type": "AND",
    "params": [
      {
        "type": "MinQuantityRule",
        "params": { "productIds": [1], "min": 5 }
      },
      {
        "type": "SumQuantityEqualsRule",
        "params": { "productIds": [1, 2], "sum": 10 }
      }
    ]
  }
}
```

Code 5 - Composite rule concrete syntax example.

This AND rule evaluates as true only if both conditions are met. More complex logic can also be composed, such as the following OR rule, that includes a nested AND block:

```
{
  "rule": {
    "type": "OR",
    "params": [
      {
        "type": "MinQuantityRule",
        "params": { "productIds": [1], "min": 5 }
      },
      {
        "type": "AND",
        "params": [
          {
            "type": "MinQuantityRule",
            "params": { "productIds": [1], "min": 2 }
          },
          {
            "type": "SumQuantityEqualsRule",
            "params": { "productIds": [1, 2], "sum": 10 }
          }
        ]
```

```
            }
          ]
        }
      }
```

Code 6 - Nested rules concrete syntax example.

This structure defines a fallback logic: the rules passes if either the first condition is satisfied or if the nested AND block passes both its inner rules. The recursive, nested nature of this DSL allows for deep and expressive logical trees.

Ultimately, the concrete syntax serves as a practical bridge between the abstract model and the real-world implementation, enabling non-developers to engage with complex logic while maintaining the system reliability and adaptability.

1.19Implementation Details

The implementation of the DSL and its supporting architecture was carried out in Java, leveraging the Spring framework for dependency injection and service orchestration. The system is composed of several modular components, each of which plays a critical role in bridging the authored DSL definitions to executable business logic.

1.19.1 Rule Factory and Runtime Discovery

The Rule Factory is the dynamic core of the implementation, responsible for discovery, registration and instantiation of rule classes. Discovery is achieved through the use of the Reflections library, which scans the classpath at startup for Java classes annotated with a custom annotation, @RuleDefinition. Each discovered class is registered in an internal registry, keyed by a string identifies corresponding to the DSL keyword (e.g., “MinQuantityRule”, “SumQuantityEqualsRule”). This design enables runtime extensibility, allowing new rule types to be added simply by creating a new annotated class without requiring any modification to the logic or recompilation of the engine.

Instantiation follows a flexible constructor-injection strategy. When a DSL campaign is parsed, its JSON representation provides a set of parameters (e.g., product identifiers, thresholds or values). These parameters are automatically mapped to the corresponding constructor arguments of the rule class. By supporting loose coupling between DSL definitions and rule constructors, the factory ensures that new business requirements can be incorporated seamlessly without modifying existing components. This approach significantly improves agility while reducing maintenance overhead.

1.19.2 Rule Context and Evaluation

At runtime all evaluations occur within a Rule Context, which encapsulates the relevant data for rule execution. The RuleContext contains:

- The products and its information for evaluation.

- Contextual metadata, like client, supplier, campaign and requester information.

The Rule Context Builder is responsible for constructing this context, fetching data dynamically from the required services. This ensures that evaluations are always performed against current operational data.

Rules implement a common Rule interface, exposing the method evaluate(RuleContext): RuleResult. Each execution produces a RuleResult, which indicates whether the rule passed, the reason for failure (if any), and the results of subordinate evaluations in case of composite rules. Composite rules such as AndRule and OrRule, evaluate lists of sub-rules recursively, enabling the hierarchical composition of complex conditions.

1.19.3 Intermediate Representation and Mapping from the DSL

Between the raw JSON DSL and the runtime classes, an intermediate representation is employed. The intermediate representation acts as a form of campaign definition, decoupling authoring syntax from runtime execution. Once a JSON DSL document is parsed, it is validated structurally (schema checks) and semantically (e.g., verifying that all referenced product ID exist). The intermediate representation is then passed to the Rule Factory, which instantiates the corresponding Java rule objects, effectively binding high-level DSL constructs to executable classes.

1.19.4 Validation Pipeline

Validation occurs at multiple levels:

- Schema validation: ensures that the DSL documents conform to the JSON schema, preventing syntactic errors.

- Semantic Validation: Confirms consistency of the parameters, such as ensuring that product IDs are valid and that numerical constraints hold true.

- Runtime Validation: During evaluation, the system verifies that rule semantics are respected in the given context.

1.20Benefits of the DSL Implementation

The implementation of the DSL showed several significant benefits that reinforce the relevance of this approach for the management of business rules and campaigns in the ePharma system.

One of the primary benefits lies on the alignment between domain knowledge and system implementation. The DSL provides a vocabulary that reflects the concepts, constraints and operations specific to the pharmaceutical ordering domain. This alignment reduces the cognitive gap between business analysts, who define the campaigns, and the software engineers, who implement them. As a result, campaigns can be expressed in a language that is

closer to the way business stakeholders' reason about rules, while remaining executable by the system.

Another benefit is the improvement in campaign maintainability and agility. The DSL enables campaigns to be updated, extended or replaced without requiring extensive modifications to core system logic. By externalizing rules from business services and encapsulating them within an independent rule engine, the implementation reduces coupling, allowing organizations to adapt campaigns quickly.

The process demonstrated the expressive power and suitability of the DSL for the domain problem in question. The language was able to capture a wide variety of commercial rules, in a form that remains both machine-executable and accessible to non-technical stakeholders. Although not all possible rules were captured, the DSL can implement almost all known case scenarios and shows promising results to implement future campaigns that weren't considered. These findings highlight the potential of DSLs to bridge the existing gap between domain expertise and system implementation, enabling business analysts to directly express campaigns without constant mediation by software engineers.

The system also enhances transparency and auditability. Since campaigns are authored in a structed form and stored in a centralized repository, it becomes possible to trace rule changes across versions, validate campaigns prior to deployment and simulate their effects. This provides organizations with the ability to justify campaign decisions, comply with regulatory demands and minimize risks of unintended outcomes.

Moreover, the validation pipeline built into the DSL architecture, which includes schema checks and contextual consistency analysis, proved essential for ensuring that authored rules were syntactically correct and semantically coherent. Runtime validation, in turn, safeguards the correct execution of rules, by verifying rule semantics and ensuring compatibility between DSL constructs and the target runtime environment. This validation process emerged as a critical enabler of trust in the system, reducing the likelihood of rule misconfiguration and unintended business outcomes.

The inclusion of governance mechanisms such as versioning, change control and publishing workflows proved invaluable in supporting the operational realities of campaign management. Campaigns are not static artifacts but evolve continuously in response to business needs. The ability to track versions and safely roll out new policies ensured that governance requirements were met without sacrificing agility.

Finally, the architecture demonstrated adaptability and extensibility. The modular structure – with components such as the Rule Factory, Evaluator Core and Rule Context Builder – allows for a straightforward addition of new rule types and the integration of external services. By decoupling rule evaluation from business services and introducing components such as the Evaluator Core, Rule Factory and Rule Context Builder, the system was able to execute campaigns efficiently while maintaining loose coupling with external services. This modularity also makes the architecture more maintainable, facilitating the addition of new rule types or integration with additional data sources. This makes the DSL implementation not only a solution to immediate business needs, but also a foundation that can evolve with future requirements.

1.21Limitations and Challenges

Despite the clear benefits, the implementation of the DSL is not without its limitations and challenges. A notable limitation is the complexity of authoring composite rules. While the DSL provides mechanisms to represent logical combinations of rules, expressing deeply nested conditions can become difficult for non-technical users. This creates a tension between expressiveness and usability. This limitation can be reduced by introducing a graphical interface that facilitates the use of the DSL.

Another challenge arises from the dependency on external services during the runtime evaluation. The Rule Context Builder relies on data fetched from multiple sources such as entity, product and user services. This introduces potential points of failure and performance bottlenecks. Ensuring the availability, responsiveness and consistency of these services is therefore critical for a reliable campaign execution but remains outside the direct control of the DSL itself.

In addition, the validation pipeline, while robust, requires ongoing maintenance. The validation of the rules must be continuously updated to reflect new domain concepts and business requirements. Similarly, runtime validation introduces overhead, which, although necessary, can add complexity to the execution workflow. Balancing thorough validation with performance efficiency is an ongoing concern.

There are also governance and adoption challenges. While the DSL provides mechanisms for versioning, change control and simulation, its effectiveness depends on organizational practices and user adoption. Business users may initially resist the shift to a DSL-based authoring of campaigns, perceiving it as a technical task. Effective training, documentation and user-friendly interfaces are essential for a smooth transition.

1.22Comparison with Existing Approaches

To assess the effectiveness of the implemented DSL-bases solution, it is necessary to contrast it with traditional approaches to business rule management commonly found in enterprise systems.

1.22.1 Hard-coded business rules in application services

In many legacy systems, commercial logic is embedded directly within the application code, typically through conditional statements scattered across multiple services. While straightforward and fast to implement initially, this approach suffers from severe drawbacks when it comes to altering the business logic, finally resulting in increased time-to-market for campaign updates. Furthermore, the tight coupling between business logic and application code creates high maintenance costs and a high risk of introducing regressions when modifying rules. In contrast, the DSL externalizes campaign logic, allowing rules to be define declaratively and evolve independently of the underlying services, thereby reducing coupling and improving maintainability.

1.22.2 Configuration tables and database-driven rules

Another widespread approach is to encode rules as configuration data stored in relational databases, often supported by lookup tables or parameterized templates. While this approach separates rules from the code, it lacks the expressive power to model complex logical compositions (e.g., nested AND/OR conditions) and is often limited to simple thresholds or static ranges. Moreover, database-driven configurations tend to become opaque as the number of entries grows, reducing traceability and usability for business stakeholders. The proposed DSL, by contrast, provides a structured but yet expressive language capable of capturing both atomic and composite constraints while preserving readability.

1.22.3 General-purpose Business Management Systems (BRMS)

Commercial solutions such as Drools, provide powerful and feature-rich environments for the rule definition and execution. These frameworks are highly extensible and include advanced features life interface engines, rule prioritisation and decision tables. However, they introduce a high level of complexity, a steep learning curve and integration overhead that may be excessive for the specific scope of ePharma's campaign configuration needs. Additionally, generic BRMS platforms may impose licensing costs and architectural constraints that conflict with the lightweight, domain-focused requirements of this thesis. By contrast, the custom DSL-based solution delivers domain-specific expressiveness and extensibility while avoiding the overhead of adopting a heavyweight general-purpose engine.

1.22.4 Visual policy builders and low-code platforms

Low-code planforms and visual authoring tools allow business stakeholders to configure workflows and rules through graphical interfaces. These tools improve usability but often lack transparency in how rules are executed at runtime, as the underlying execution semantics are hidden within proprietary engines. This reduces auditability and makes debugging difficult. The proposed DSL solution explicitly defines both the syntax and semantics of rules, ensuring transparency and traceability. Moreover, while the current implementation provides a JSON-bases DSL, its design is compatible with a future graphical interface that could generate the underlying DSL automatically, thus combining usability with transparency.

Verification and Validation

The verification and validation of the DSL and its supporting rule engine were carried out through a systematic testing strategy designed to ensure both functional correctness and robustness. Given the critical role of the rule engine in defining and enforcing campaigns, the evaluation was performed across multiple levels: unit tests, mutation tests, and integration tests. These levels collectively provide evidence of correctness for isolated components, resilience of the test suite against regressions, and the correct orchestration of rules in realistic business scenarios.

The tests presented in this section are representative examples. In practice, a more extensive suite was implemented. The following subsections outline the methodology, illustrate sample tests, and summarise the overall coverage obtained.

1.23Unit Tests

Unit testing is the process where tests are made to the smallest functional unit of code (AWS, 2025). It ensures that each rule implementation behaves as expected in isolation, under both valid and invalid conditions. These tests are critical for this project because:

- Rules are self-contained modules that can be tested independently of the rest of the system.
- The DSL is user-facing, meaning that invalid inputs must be rejected.

Since rules must implement the Rule interface and expose the evaluate(RuleContext) method, unit tests are focused on two main aspects:

- Verification of parameters during rule construction: ensuring the rule rejects malformed or incomplete definitions.
- Functional verification: ensuring the evaluate method produces the expected RuleResult when applied to controlled contexts.

1.23.1 Verification

The following unit test verifies that invalid parameter structures, result in exceptions at construction time. For example, the MinQuantityRule requires a list of product IDs and an integer minimum value.

```
@Test
void testInvalidParametersForMinQuantityRule() {
    Map<String, Object> invalidParams = Map.of(
        "productIds", "notAList", // should be List<Long>
        "min", "five"             // should be Integer
    );
```

```
        assertThrows(IllegalArgumentException.class,
            () -> new MinQuantityRule(invalidParams));
    }
```

Code 7 - Unit test example

This ensures that malformed DSL definitions cannot propagate into the runtime environment.

1.23.2 Functional Verification

This unit test asserts that the rule passes when the defined thresholds are met:

```
    @Test
    void testMinQuantityRulePassesWhenThresholdMet() {
        Map<String, Object> params = Map.of(
            "productIds", List.of(1L),
            "min", 5
        );
        MinQuantityRule rule = new MinQuantityRule(params);

        Product product = new Product(1L, "Aspirin", 10);
        RuleContext context = RuleContext.builder()
            .products(List.of(product))
            .build();

        RuleResult result = rule.evaluate(context);

        assertTrue(result.isPassed());
        assertTrue(result.getFailureReasons().isEmpty());
    }
```

Code 8 – Functional verification unit test example.

Complementary tests check failure conditions, ensuring that when the product has fewer units than required, the rule fails with a clear failure reason.

1.23.3 Coverage

Unit tests were implemented not only for MinQuantityRule, but also for other rules. Additional tests were added for boundary cases, such as empty products lists, null contexts, or invalid parameter values.

Table 2 - Unit tests.

Test ID	Rule	Category	Scenario	Description	Expected outcome
UT-01	MinQuantityRule	Validation	Missing productIds	productIds absent	Exception thrown
UT-02	MinQuantityRule	Validation	productIds wrong type	productIds not a list	Exception thrown
UT-03	MinQuantityRule	Validation	Missing min	Min field absent	Exception thrown

UT-04	MinQuantityRule	Validation	Min wrong type	Min is a string	Exception thrown
UT-05	MinQuantityRule	Execution	Valid quantity	Product quantity is 10 min is 5	Pass
UT-06	MinQuantityRule	Execution	Below min	Product quantity is 10 min is 20	Fail, message includes required threshold
UT-07	MinQuantityRule	Execution	Boundary	Product quantity is 10 min is 10	Pass
UT-08	MinQuantityRule	Execution	Product missing	Product not in context	Fail, with message
UT-09	MinQuantityRule	Execution	Multi-product	Product 1, 2 and 3 quantity ≥ min	Pass
UT-10	MinQuantityRule	Execution	Multi-product fail	Product 2 below min	Fail, with specific producto ID in message
UT-11	AndRule	Validation	Invalid subrule type	Non-list input	Exception thrown
UT-12	AndRule	Execution	All pass	Subrules all true	Pass
UT-13	AndRule	Execution	One fails	At least one subrule false	Fail
UT-14	AndRule	Execution	All fail	All subrules false	Fail

1.24Mutation Tests

Unit tests establish functional correctness, but they do not evaluate its capability to detect subtle regressions. Mutation testing addresses this by automatically injecting small faults (mutants) into the code and evaluating whether the existing tests “kill” these mutants (Chaurasia, 2014). The objectives were to access the fault-detection power of the test suit and identify gaps in assertions.

Mutation testing was performed using PIT (Pitest), integrated into the Maven build. PIT was configured to target the rule implementations (atomic and composite) that execute business logic and to apply a comprehensive mutator set (PITest, 2025). The PIT version used was 1.15.8 and reports were produced in HTML/XML/CSV automatically by the system. The mutators used were:

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS, VOID_METHOD_CALLS

The first level of analysis considers the overall mutation testing performance, this provides a global picture of the quality and fault-detection power of the tests.

Table 3 - Mutation tests results.

Line coverage	Mutation coverage	Test strength
97% (133/137)	94% (29/31)	100% (29/29)

As shown in Table 3, the overall results are highly positive, with a 97% line coverage and 94% mutation coverage. This demonstrates both strong structural coverage and a very high capability of killing injected mutants. The 100% test strength indicates that all covered mutants were consistently detected by at least one test, which means the tests are both broad and robust.

To complement the general metrics, PIT also provides a breakdown per class (Table 4), which enables a more fine-grained view of how well each rule type is tested. This detail is particularly important since each rule encapsulates different business logic and thus has different potential problems.

Table 4 - Mutation tests class results.

Class	Line coverage	Mutation coverage	Test strength
AndRule	100% (29/29)	100% (4/4)	100% (4/4)
OrRule	100% (30/30)	100% (4/4)	100% (4/4)
MinQuantityRule	93% (37/40)	92% (11/12)	100% (11/11)
SumQuantityEqualsRule	97% (37/38)	91% (10/11)	100% (10/10)

From this class-level breakdown, it can be observed that both composite rules achieved a 100% line and mutation coverage, reflecting their relatively simpler evaluation semantics and the

comprehensive tests applied. The atomic rules also present high scores, though with a small number of surviving mutants.

These results indicate that while the tests are already highly reliable, improvements in specific scenarios could push coverage towards a complete 100%. To provide further clarity on the types of mutants generated and the way tests successfully eliminated them, the following examples are highlighted:

1. Conditional boundary – in MinQuantityRule the code product.getQuantity() <= min would be mutated to product.getQuantity() < min, that would fail when the quantity = min. This mutation was covered by a boundary test that ensures quantity = min passes, thus killing the mutant.

2. Equality flipped – in SumQuantityEqualsRule, the code actualSum == expectedSum was mutated to actualSum != expectedSum and was covered by a test where product quantities sum exactly to the expected value, so the mutated code would fail.

These examples not only illustrate the type of faults that mutation testing introduces, but also the ability of the current tests to detect them. The combination of near-complete mutation coverage and a high-test strength confirms that the validation pipeline is rigorous and resilient, offering strong guarantees against regressions.

1.25Integration Tests

Integration tests is a software-testing approach in which various application components are tested to evaluate how well they work together (Powel and Smalley, 2025). In this case, they verify that multiple rules can be composed and evaluated together in realistic business scenarios. They ensure that the DSL definitions, rule factory and evaluation logic interact correctly. Campaign rarely consists of a single rule and the correct orchestration of composite rules is critical. Integration tests provide confidence that the DSL-defined campaigns behave as intended in real-world contexts.

The following DSL campaign required that:

1. Product 1 has at least 5 units.

2. The total quantity of the products 1 and 2 is 10.

```
{
  "rule": {
    "type": "AND",
    "params": [
      { "type": "MinQuantityRule", "params": { "productIds": [1], "min": 5
} },
      { "type": "SumQuantityEqualsRule", "params": { "productIds": [1, 2],
 "sum": 10 } }
    ]
  }
}
```

Code 9 - Integration test DSL example.

The corresponding integration test confirms that this composite rule passes under the correct conditions:

```
@Test
void testAndRuleIntegration() {
    Map<String, Object> minQuantityParams = Map.of(
        "type", "MinQuantityRule",
        "params", Map.of("productIds", List.of(1L), "min", 5)
    );
    Map<String, Object> sumQuantityParams = Map.of(
        "type", "SumQuantityEqualsRule",
        "params", Map.of("productIds", List.of(1L, 2L), "sum", 10)
    );

    AndRule andRule = new AndRule(List.of(minQuantityParams,
sumQuantityParams));

    Product p1 = new Product(1L, "ProductA", 6);
    Product p2 = new Product(2L, "ProductB", 4);
    RuleContext context = RuleContext.builder()
        .products(List.of(p1, p2))
        .build();

    RuleResult result = andRule.evaluate(context);

    assertTrue(result.isPassed());
}
```

Code 10 - Integration test example.

Other integration tests verified nested OR rules, combined AND/OR logic and failure scenarios (Table 5).

Table 5 - Integration tests.

Test ID	Scenario	Description	Expected Outcome
IT-01	Simple AND	AND(Min, Sum), both true	Pass
IT-02	Simple AND fail	AND(Min, Sum), one false	Fail
IT-03	Simple OR	OR(Min, Sum), one true	Pass
IT-04	Simple OR fail	OR(Min, Sum), both false	Fail
IT-05	Nested AND in OR	OR(AND(Min, Sum), Min)	Pass if inner AND or Min passes
IT-06	Nested OR in AND	AND(OR(Min, Sum), Min)	Pass only if both correct
IT-07	Invalid DSL structure	Missing fields	Validation error
IT-08	Invalid product reference	Product not in context	Fail
IT-09	Short-circuit OR	First passes, second skipped	Pass

IT-10	Short-circuit AND	First fails, second skipped	Fail

These tests confirm not only the correctness of composite rule evaluation but also the robustness of the engine in handling invalid structures, non-existent references and short-circuit optimisations. Together, they provide confidence that the DSL-defined campaigns function as intended in realistic business contexts.

Conclusions

This dissertation addressed the structural and operational limitations of the campaign configuration process in the ePharma platform. The work proposed, designed and implemented a Domain-Specific Language (DSL) and rule engine, with the purpose of decoupling campaign logic from the application's codebase, thereby enabling a more flexible, maintainable, adaptable and user-oriented approach to campaign management in the pharmaceutical sector.

1.26Summary of Contributions

The first contribution of this work was the formalisation of the problem domain. Through systematic domain analysis, a comprehensive model of campaigns, products, requesters, deliveries and their interdependencies. This domain not only clarifies implicit business knowledge but also provided the conceptual foundation for the design of the DSL.

Second, the dissertation introduces a DSL tailored to the pharmaceutical campaign configuration context, defined in terms of its abstract syntax, concrete syntax and formal semantics. The DSL encapsulates commercial rules as composable atomic and composite constructs, enabling for a more expressive campaign definition while maintaining readability for business stakeholders.

Third, a rule engine was implemented, comprising components such as the RuleFactory, RuleContext and Evaluator This architecture operationalises the DSL by providing mechanisms for dynamic instantiation of rules, real-time validation pipeline, semantic analysis and runtime verification, which ensures both correctness and trustworthiness of the authored campaigns.

Finally, the thesis delivered a prototype integrated into the ePharma platform, validated through unit, mutation and integration testing. The experimental evaluation confirmed that the approach is both technically feasible and robust, with high test coverage, strong fault detection capacity and reliable behaviour in realistic business scenarios.

1.27Critical Reflection

The results of this work are broadly positive. The DSL showed improvements in maintainability by externalising campaign logic from the Orders functionality, reducing the dependency on developers for routine configuration changes. It also increased agility, as campaigns can now be updated more rapidly and with reduced risk of regression. The ability to version campaigns and simulate their effects prior to deployment directly addresses the possibility of configuration errors, providing greater transparency and auditability. Moreover, the modularity of the architecture confirms its extensibility, as new rules can be introduced without modifying the engine core.

However, the work showed some limitations. A notable challenge is in usability and user adoption: while the JSON-based DSL achieves human readability, the expression of deeply

nested composite rules may still be inaccessible to non-technical users and their opening to adoption. This reflects a tension between expressive power and ease of use. Although a graphical interface can be envisioned, it remains unimplemented and is essential to achieve empowerment of business analysts.

Another limitation is the dependency on external services during runtime evaluation. The correctness and performance of rule execution rely on the availability and responsiveness of product, entity, user and other services. While this integration ensures up-to-date contextual data, it also introduces potential failure points outside the direct scope of the DSL-based evaluation engine/system.

Finally, the approach to validation and governance, while rigorous, demands ongoing maintenance. The introduction of new domain concepts or evolving business requirements will require ongoing extension of validation schemas, semantic rules and tests. Thus, sustaining the solution in the long term depends on disciplined practices and continuous investment.

1.28Future Work

Several improvements exist for extending and strengthening the contribution of this thesis.

- Graphical campaign authoring tools – the most pressing future direction is the development of a user interface that abstracts the JSON representation into visual constructs (e.g. drag-and-drop builders, forms). This would reduce the cognitive barrier for business analysts and fully realise the DSL's potential for campaign authoring.

- Advanced validation and simulation mechanisms – while the current validation pipeline ensures structural and semantic correctness, future work could extend it with domain-aware simulations. A "what-if" simulator would allow stakeholders to run campaigns against historical data or synthetic data, granting the option to visualize the outcome. This would not only prevent misconfigurations but also support decision-making by observing the results of each campaign, thereby increasing confidence in campaign correctness.

- Performance optimisation and scalability of the rule engine – although scalability wasn't the prime focus of this dissertation, as campaigns grow in complexity and the number of clients increases, the evaluation workload will rise substantially. A key direction for future work is to explore strategies for optimization, like rule caching and parallel execution across distributed services. These improvements would ensure that even highly complex campaigns can be evaluated reliably.

1.29Final Remarks

This thesis demonstrates that a DSL-based approach to campaign configuration is both viable and advantageous in the pharmaceutical orders software domain. By providing a formalised yet accessible mechanism for defining campaign logic, supported by a robust execution framework, the solution significantly improves maintainability, adaptability and usability of the ePharma platform.

At the same time, the work highlights the inherent trade-offs in DSL design, particularly between expressive capacity and usability, as well as the ongoing governance challenges of sustaining such a system in a dynamic business environment. Nevertheless, the contributions presented here constitute a solid foundation upon which ePharma can evolve its campaign configuration system into a more agile, user-empowered, and future-proof solution.

Bibliography

(Artho et al., 2015)	Artho, C., Klaus Havelund, Kumar, R. and Yamagata, Y. (2015). Domain-Specific Languages with Scala. Lecture Notes in Computer Science, [online] pp.1–16. doi:https://doi.org/10.1007/978-3-319-25423-4_1.
(AWS, 2025)	AWS (2025) 'What is Unit Testing?', AWS. Available at: https://aws.amazon.com/what-is/unit-testing/ (Accessed: 20 September 2025).
(Berzak, 2023)	Berzak, D. (2671). Embedded Domain Specific Language: A Streamlined Approach for Framework Abstraction. [online] Available at: https://atlarge-research.com/pdfs/2023-dberzak-bsc_thesis.pdf [Accessed 12 June 2025].
(Borum, 2022)	Borum, H. S. (2022) The Design and Implementation of the Management Action Language and the Life Cycle of Other DSLs. PhD thesis, IT University of Copenhagen. Available at: https://pure.itu.dk/ws/portalfiles/portal/109068125/PhD_Thesis_Final_version_Holger_Stadel_Borum.pdf (Accessed: 5 September 2025).
(Chaurasia, 2014)	Chaurasia, P. (2014) 'Mutation testing: a review', Journal of Global Research in Computer Sciences, [online] 1 February. Available at: https://www.researchgate.net/publication/361016868_MUTATION_TESTING_A_REVIEW (Accessed: 15 September 2025).
(Christensen, 2003)	Christensen, N. H. (2003) *Domain-Specific Languages in Software Development and the Relation to Partial Evaluation*. PhD thesis, University of Copenhagen. Available at: https://di.ku.dk/forskning/Publikationer/tekniske_rapporter/tekniske-rapporter-2003/03-09.pdf
(Drools, 2013)	Drools (2013) 'Chapter 6: The Rule IDE (Eclipse)', in *Drools Expert Documentation*, version 6.0.0.CR1. Available at: https://docs.drools.org/6.0.0.CR1/drools-expert-docs/html/ch06.html (Accessed: 15 September 2025).
(Drools, 2025)	Drools (2025) Drools - Business Rules Management System (Java™, Open Source). Available at: https://www.drools.org [Accessed: 16 June 2025].
(ePharma 2024)	ePharma. Available at: https://www.epharma.pt/ aboutUs.html (Accessed: 26 November 2024).
(Fowler and Parsons, 2010)	Fowler, M. (2010). Domain-Specific Languages. Pearson Education.
(Fowler, 2005)	Fowler, M. (2005) 'Language Workbenches: The Killer-App for Domain Specific Languages?', MartinFowler.com. Available at: https://martinfowler.com/articles/languageWorkbench.html#InternalDsl [Accessed: 26 June 2025].
(Fowler, 2005)	Fowler, M. (2005) 'A Language Workbench in Action – MPS', MartinFowler.com. Available at: https://martinfowler.com/articles/mpsAgree.html (Accessed: 5 September 2025).

(Fowler, 2006) Fowler, M. (2006) 'Internal DSL Style', MartinFowler.com. Available at: https://martinfowler.com/bliki/InternalDslStyle.html [Accessed: 26 June 2025].

(Fowler, 2008) Fowler, M. (2008) 'DSL Q & A', MartinFowler.com. Available at: https://martinfowler.com/bliki/DslQandA.html [Accessed: 26 June 2025].

(Fowler, 2010) Fowler, M. and Parsons, R. (2010) *Domain-Specific Languages*. Addison-Wesley Professional.

(Gomez-Vazquez and Cabot, 2025) Gomez-Vazquez, M. and Cabot, J. (2025) 'Towards a DSL to Formalize Multimodal Requirements', arXiv preprint. Available at: https://arxiv.org/html/2508.14631v1 [Accessed: 20 July 2025].

(Gray et al., 2008) Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M. and Tolvanen, J.-P. (2008) 'DSLs: The good, the bad, and the ugly', *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA Companion 2008)*, pp. 791–794. doi:10.1145/1449814.1449863.

(Groff and Weinberg, 2009) Groff, J. R. and Weinberg, P. N. (2009) *SQL: The Complete Reference* (3rd edn). McGraw-Hill.

(Hermans, Pinzger and van Deursen, 2009) Hermans, F., Pinzger, M. and van Deursen, A. (2009) 'Domain-Specific Languages in Practice: A User Study on the Success Factors', in A. Schürr and B. Selic (eds) *Model Driven Engineering Languages and Systems (MODELS 2009)*, Lecture Notes in Computer Science, vol. 5795, Springer, pp. 423-437. doi:10.1007/978-3-642-04425-0_33.

(HGData, 2025) HGData (no date) 'Drools', HGData Discovery. Available at: https://discovery.hgdata.com/product/drools (Accessed: 15 September 2025).

(Hudge, 2024) Hudge, S. (2024) 'Layers in software architecture', Medium. Available at: https://medium.com/@sagar.hudge/layers-in-software-architecture-c8cc16329ff6 [Accessed: 5 July 2025].

(JetBrains, 2024) JetBrains (2024) 'Refactoring', MPS Documentation, JetBrains. Available at: https://www.jetbrains.com/help/mps/mps-refactoring.html (Accessed: 30 June 2025).

(JetBrains, 2025) JetBrains (2025) Meta Programming System: Design your own Domain Specific Language with full development environment. Available at: https://www.jetbrains.com/opensource/mps/ [Accessed: 17 June 2025].

(JetBrains, 2025a) JetBrains (2025a) 'Projection Editor', MPS Concepts. Available at: https://www.jetbrains.com/mps/concepts/#projection-editor (Accessed: 15 September 2025).

(Keshishzadeh and Mooij, 2014) Keshishzadeh, S. and Mooij, A. J. (2014) 'Formalizing DSL Semantics for Reasoning and Conformance Testing', in Software Engineering and Formal Methods (SEFM 2014), Lecture Notes in Computer Science, pp. 81–95. Available at: https://doi.org/10.1007/978-3-319-10431-7_7.

(Kosar et al., 2010)	Kosar, T., Oliveira, N., Mernik, M., Varanda Pereira, M. J., Črepinšek, M., da Cruz, D. and Henriques, P. R. (2010) 'Comparing General-Purpose and Domain-Specific Languages: An Empirical Study', Computer Science and Information Systems, 7(2), pp. 247-264. doi:10.2298/CSIS1002247K.
(Kumar et al., 2011).	Kumar, N., et al. (2011) 'Rule based programming with Drools', International Journal of Computer Science and Information Technologies (IJCSIT), 2(3), pp. 1121–1126. Available at: https://www.ijcsit.com/docs/Volume%202/vol2issue3/ijcsit2011020335.pdf [Accessed: 16 June 2025].
(Larman, 2004)	Larman, C. (2004) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd edn). Prentice Hall.
(Mernik, Heering and Sloane, 2005)	Mernik, M., Heering, J. and Sloane, A.M. (2005). When and how to develop domain-specific languages. ACM Computing Surveys, 37(4), pp.316–344. doi:https://doi.org/10.1145/1118890.1118892.
(Meyers et al., 2017)	Yentl Van Tendeloo, Mierlo, S.V., Meyers, B. and Vangheluwe, H. (2017). Concrete syntax: a multi-paradigm modelling approach. pp.182–193. doi:https://doi.org/10.1145/3136014.3136017.
(Microsoft, 2025)	Microsoft (2025) 'Azure Database for MySQL – Flexible Server overview', Microsoft Learn. Available at: https://learn.microsoft.com/en-us/azure/mysql/flexible-server/overview (Accessed: 11 September 2025).
(MongoDB, 2025)	MongoDB (2025) 'Reduce the Size of Large Documents', MongoDB Atlas Documentation. Available at: https://www.mongodb.com/docs/atlas/schema-suggestions/reduce-document-size/ (Accessed: 1 September 2025).
(Paczona et al., 2024)	Paczona, M., et al. (2024) 'Increase development productivity by domain-specific modeling methods', *Journal of Systems and Software*. doi:10.1016/S0169023X23001234. Available at: https://www.sciencedirect.com/science/article/pii/S0169023X23001234 (Accessed: 5 September 2025).
(Pawar, 2025)	Pawar, A. (2025) 'Building Internal DSLs in Your Favorite Programming Language', softAai Blogs (Medium). Available at: https://medium.com/softaai-blogs/building-internal-dsls-in-your-favorite-programming-language-beb329021c7e [Accessed: 26 June 2025].
(PITest, 2025)	PITest (2025) 'Real world mutation testing — PIT', Available at: https://pitest.org (Accessed: 15 September 2025).
(Powell and Smalley)	Powell, P. and Smalley, I. (2025) 'What is integration testing?', IBM Think. Available at: https://www.ibm.com/think/topics/integration-testing (Accessed: 20 September 2025).
(Powell, 2010)	Powell, T. A. (2010) *HTML & CSS: The Complete Reference* (5th edn). McGraw-Hill.

(Quality Manageme nt, 2018)	Quality Management (2018) 'FURPS model', Quality Management (WordPress). Available at: https://qualitymanagement387741893.wordpress.com/2018/07/03/furps-model/ (Accessed: 1 September 2025).
(Réveillere et al., 2000)	Réveillere, L., Merillon, F., Consel, C., Marlet, R. and Muller, G. (2000) 'A DSL Approach to Improve Productivity and Safety in Device Drivers Development', IEEE/ACM Symposium on Automated Software Engineering (ASE '00). Available at: https://who.paris.inria.fr/Gilles.Muller/papers/ase00-devil.pdf (Accessed: 5 September 2025).
(Richardson, 2025)	Richardson, C. (no date) 'Shared database', Microservices.io. Available at: https://microservices.io/patterns/data/shared-database.html (Accessed: 11 September 2025).
(Ruiz and Bay, 2008)	Ruiz, A. and Bay, J. (2008) 'An Approach to Internal Domain-Specific Languages in Java', InfoQ. Available at: https://www.infoq.com/articles/internal-dsls-java/ (Accessed: 1 September 2025).
(Schneller, 2010)	Schneller, D. (2010) 'Code Generation With Xtext', DZone. Available at: https://dzone.com/articles/code-generation-xtext (Accessed: [6 September 2025]).
(Sharma, 2024)	Sharma, K. (2024) 'The role of domain-specific languages (DSLs) in tailoring solutions for specific industries', Skillions Technologies. Available at: https://skillions.in/the-role-of-domain-specific-languages-dsls-in-tailoring-solutions-for-specific-industries/ [Accessed: 14 June 2025].
(Spinellis, 2001)	Spinellis, D. (2001). Notable design patterns for domain-specific languages. Journal of Systems and Software, 56(1), pp.91–99. doi:https://doi.org/10.1016/s0164-1212(00)00089-3.
(Swift, 2024)	Swift, D. (2024) 'Leveraging Service Mesh for Java Microservices Scalability and Security', Springfuse. Available at: https://www.springfuse.com/service-mesh-for-scalable-microservices/ [Accessed: 25 June 2025].
(TLVTech 2025)	Exploring Domain-Specific Languages: A Practical Overview. Available at: https://www.tlvtech.io/post/understanding-value-of-domain-specific-languages [Accessed: 12 June 2025].
(Voelter and Lisson, 2014)	Voelter, M. and Lisson, S. (2014) 'Supporting Diverse Notations in MPS' Projectional Editor', in CEUR Workshop Proceedings, Vol. 1236. Available at: https://ceur-ws.org/Vol-1236/paper-03.pdf (Accessed: 6 September 2025).
(Voelter et al., 2010)	Voelter, M., Ratiu, D., Schaetz, B. and Kolb, B. (2010) 'mbeddr: an Extensible MPS-based Programming Language and IDE for Embedded Systems', JetBrains. Available at: https://resources.jetbrains.com/storage/products/mps/docs/mbeddr-mps-casestudy.pdf [Accessed: 17 June 2025].
(Xtend, 2025)	Xtext (2025) 'Configuration', Xtext Documentation. Available at: https://eclipse.dev/Xtext/documentation/302_configuration.html (Accessed: 6 September 2025).
(Xtext, 2025)	Xtext (2025) Xtext – Language Engineering Made Easy! Available at: https://eclipse.dev/Xtext/ [Accessed: 24 June 2025].
(Xtext, 2025)	Xtext (2025) Xtext – Language Engineering Made Easy! Available at: https://eclipse.dev/Xtext/ (Accessed: 15 September 2025).

(Zaytsev and Tomassetti, 2020)	Tomassetti, F. and Zaytsev, V. (n.d.). Reflections on the Lack of Adoption of Domain Specific Languages. [online] Available at: https://grammarware.net/text/2020/dsl-adoption.pdf [Accessed: 16 June 2025].

Appendices

Appendix A: Additional Configuration Dimensions

- Minimum PVA NET value: the minimum PVA NET value the campaign needs to have to be valid.
- Maximum PVA NET value: the maximum PVA NET value the campaign can have to be valid.
- Minimum PVP value: the minimum PVP value the campaign needs to have to be valid.
- Maximum PVP value: the maximum PVP value the campaign can have to be valid.
- Minimum PVF value: the minimum PVP value the campaign needs to have to be valid.
- Maximum PVF value: the maximum PVP value the campaign can have to be valid.
- Minimum discount value: the minimum discount value the campaign needs to have to be valid.
- Maximum discount value: the maximum discount value the campaign can have to be valid.
- Minimum bonus units: the minimum bonus units the campaign needs to have to be valid.
- Maximum bonus units: the maximum bonus units the campaign can have to be valid.
- Minimum Delivery Dates: the minimum number of delivery dates a campaign must have.
- Maximum Delivery Dates: the maximum number of delivery dates a campaign can have.
- Product Family Minimum Quantity: the minimum quantity of product families a campaign must have to be valid.
- Product Family Maximum. Quantity: the maximum quantity of product families a campaign can have to be valid.
- Bonus Manual Override: defines if a campaign allows the edition of bonus, overriding the defined values.
- Discount Manual Override: defines if a campaign allows the edition of discounts, overriding the defined values.
- Mandatory Superior Validation: defines if a campaign must be always validated by a superior.

Appendix B: Additional Configuration Product Dimensions

- Discount ranges by PVA: The discount ranges by total PVA value of the product.
- Discount ranges by PVP: The discount ranges by total PVP value of the product.
- Discount ranges by requester tier: the discount ranges by requester tier.
- Bonus ranges by PVA: the bonus ranges by total PVA value of the product.
- Bonus Ranges by PVP: the bonus ranges by total PVP value of the product
- Bonus multiplier: the bonus multiplier of the product.
- Bonus range multipliers: the bonus multiplier by unit's quantity of the product.
- Minimum quantity by month: the minimum product unit's quantity by month the product needs to be valid.
- Maximum quantity by month: the maximum product unit's quantity by month the product needs to be valid.
- Minimum quantity by delivery: the minimum product unit's quantity by delivery the product needs to be valid.
- Maximum quantity by delivery: the maximum product unit's quantity by delivery the product needs to be valid.
- Minimum PVA NET value: the minimum PVA NET value the product needs to have to be valid.
- Maximum PVA NET value: the maximum PVA NET value the product can have to be valid.
- Minimum PVP value: the minimum PVP value the product needs to have to be valid.
- Maximum PVP value: the maximum PVP value the product can have to be valid.
- Minimum PVF value: the minimum PVP value the product needs to have to be valid.
- Maximum PVF value: the maximum PVP value the product can have to be valid.