INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




# Editor de código para objetos de negócio

**ÓSCAR MARQUES MENDES FOLHA**
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

# Editor de código para objetos de negócio

**Óscar Marques Mendes Folha**

**Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software**

**Advisor: Paulo Gandra de Sousa**

Porto, Junho 2025

# Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 28 de junho de 2025

# Resumo

Este documento aborda os desafios e benefícios associados à implementação de sistemas de edição e validação de código em tempo real em ambientes web, com particular foco na integração de estruturas e modelos de dados. Este trabalho surge da necessidade de colmatar uma limitação concreta da PMWeb, uma aplicação web desenvolvida para o setor segurador, que apesar de permitir uma gestão eficaz de dados versionados, não oferece funcionalidades de edição e validação de fórmulas em tempo real. Esta lacuna obriga ainda à utilização de sistemas legacy para configurar produtos de seguro, criando obstáculos à autonomia e eficiência dos utilizadores.

Com base na evolução da própria PMWeb, este projeto teve como objetivo investigar formas de integrar capacidades de edição de código com validação sintaxe em tempo real, suporte a múltiplos utilizadores e ligação direta a modelos de dados. Durante a revisão do estado da arte foram identificadas várias limitações nas soluções atuais, mas também abordagens promissoras, como o uso do Language Server Protocol (LSP), o editor Monaco e bases de dados com controlo de versões como o Dolt. A solução aqui apresentada resulta da combinação destas tecnologias, permitindo criar um sistema mais flexível e colaborativo, adaptado à realidade de ambientes intensivos em dados como os das seguradoras.

**Palavras-chave**: validação de sintaxe em tempo real, autocompletar e modelo de objetos de negócio.

# Abstract

This document addresses the challenges and benefits associated with implementing real-time code editing and validation systems in web-based environments, with a particular focus on the integration of data structures and models. This project was motivated by a clear limitation in PMWeb, a web tool used by insurance companies to manage data with version control. Although PMWeb works well for handling structural updates, it doesn't yet support writing or validating formulas directly in the platform. Because of this, many users still depend on older desktop software, which limits their ability to work independently and adapt quickly.

The goal of this work is to address that gap by bringing formula editing into the browser. This includes real-time syntax checking, helpful suggestions during editing, and direct integration with the system's structured data models.The research involved reviewing current technologies and identifying shortcomings, while also recognizing promising tools such as the Language Server Protocol (LSP), Monaco Editor, and version-controlled databases like Dolt. The proposed solution combines these technologies into a single architecture that improves usability, encourages collaboration, and helps bring PMWeb closer to becoming a fully self-contained platform for insurance product configuration.

**Keywords**: real-time syntax validation, autocomplete, business object model.

# Acknowledgments

I would like to express my deepest gratitude to Professor Paulo Gandra de Sousa for his invaluable guidance and support throughout the completion of this work. His expertise and dedication were essential to the development of this project.

I extend my thanks to msg life for the trust placed in me and for the opportunity to take on this challenge, as well as to my colleagues Hugo Ferreira, Joel Silva, João Brito, João Có, and André Gonçalves for their support, collaboration, and motivation along the way.

I am also profoundly grateful to my classmates Rafael Faísca, Rogério Sousa, Rafael Oliveira, and Nuno Marmeleiro for their camaraderie and encouragement during the more challenging moments.

Lastly, heartfelt thanks to my family for their unconditional support, patience, and strength, which have always been a source of inspiration, and to the Instituto Superior de Engenharia do Porto (ISEP) for providing the resources and environment necessary to complete this dissertation.

# Table of Contents

# List of Figures

# List of Tables

# Acronyms and Symbols

## List of Acronyms

**BOM**: Business Object Model

**DSL**: Domain-Specific Language

**EC**: Exclusion Criteria

**IC**: Inclusion Criteria

**IDE**: Integrated Development Environment

**ISEP**: Instituto Superior de Engenharia do Porto

**JSON**: JavaScript Object Notation

**LSP**: Language Server Protocol

**OCL**: Object Constraint Language

**PM**: Product Machine – msg life application

**PMWeb:** Product Machine Web – msg life application

**PRISMA**: Preferred Reporting Items for Systematic Reviews and Meta-Analyses

**RQ**: Research Question

**RPC:** Remote Procedure Call

# 1 Introduction

## 1.1 Context

The insurance industry increasingly demands adaptable and efficient digital solutions to model and manage the structure and behavior of insurance products. *PM* (Product Machine) a msg life product is a comprehensive, desktop-based application that has long served as the backbone of insurance product modeling. It enables insurers to define, customize, and manage the structural and behavioral aspects of their products through a modular framework, providing tools for end-to-end product lifecycle management.

**Main Modules of the PM System**:

1. **Product Definition Module**:
   - This module plays a central role in defining the structure of insurance products. It enables users to model core elements such as components, features, and business rules. Attributes, properties, and relationships between product elements, including for example, packages, coverages, and contracts and they can all be configured in a flexible and structured way.
2. **Business Object Model (BOM)**:
   - Acts as a central framework outlining relationships between business objects;
   - Key components include:
     - **Packages**: Groupings of related products or features;
     - **Contracts**: Terms and conditions governing products;
     - **Roles**: Roles performed within the scope of a contract;
     - **Experiences**: User interactions with products based on role and context.

3. **Customization Module**:
    - Allows tailoring of product definitions for client-specific needs;
    - Features include parent-child relationships, rules for variations, and arguments for dynamic behaviors.

4. **Change Management Module**:
    - Tracks and implement updates to product definitions;
    - Includes version control, change sets, and lineups (sets of grouped changes for deployment).

5. **Testing and Validation Module**:
    - Ensures product definitions function correctly before deployment;
    - Tools like CETA and test accelerators validate changes and enforce compliance with business rules.

6. **Data Management Module**:
    - Handles underlying data structures supporting product configurations;
    - Allows management of multi-criteria data tables for flexible modeling.

7. **User Interface Module**:
    - Provides an interactive front-end for creating, managing, and navigating product definitions.

8. **Integration Module**:
    - The integration module ensures smooth communication between the PM system and external platforms. Its main purpose is to keep data synchronized and consistent across systems, allowing for efficient data exchange and minimizing the risk of mismatches or errors.

Although PM has proven to be stable and dependable over the years, its desktop-based architecture introduces some practical limitations. Users need to have this application installed locally, which makes deployments harder to do and the data is not shared by all users, making collaborative work harder to achieve. These limitations led to the creation of PMWeb, a web-based version of the system designed to make it easier to collaborate, scale, and move away from the constraints of a desktop environment.

**Key Features of PMWeb**:

1) Initially focused on *master data management*, managing critical elements such as:
   - Postal codes;
   - Risk factors;
   - Documents associated with processes.

2) Incorporates *versioned data management*, enabling insurers to:
   - Maintain historical records of changes for compliance and auditing;
   - Experiment with data branches without affecting live data;
   - Collaborate on isolated versions of datasets and merge updates as needed.

However, PMWeb currently lacks advanced functionalities for behavioral modeling, particularly in areas like formula creation and validation. These limitations hinder its ability to fully replace PM and provide a unified solution for managing both the structure and behavior of insurance products.

One of the most critical gaps lies in the use of *formulas*, which are essential for:

- **Validations**: Ensuring data integrity and compliance;
- **Conditional Component Composition**: Dynamically assembling product components;
- **Premium Calculations**: Defining rules for calculating insurance costs based on configurable parameters.

Despite PMWeb's advancements in data versioning and master data management, the lack of integrated behavioral modeling functionalities, particularly formula creation and validation, limits its potential as a full replacement for PM.

## 1.2 Problem description

The absence of a formula editor in PMWeb creates significant gaps in its ability to handle the behavioral modeling of insurance products. In its current state, PMWeb is limited to managing master data tables, forcing clients to rely on the legacy PM system for advanced functionalities like formula definition and validation. This dual-system dependency creates inefficiencies and fragmentation for users who need to seamlessly manage both the structure and behavior of insurance products in a unified platform.

The PM system includes strong support for defining business logic, especially through formulas written in the Object Constraint Language (OCL). These formulas are used throughout the platform and serve several important roles. For example, they help validate the Business Object

Model (BOM) by checking that attributes and relationships between entities are consistent with the business rules defined by the product. They are also used in the Change Management Module to ensure that any updates to product definitions follow the required conditions, which helps prevent invalid or inconsistent configurations from being saved. In the Customization Module, formulas allow for dynamic behaviour and conditional logic, making it easier to make products to the specific needs of each client. During testing, they also play a key role in ensuring that all business rules are followed before anything goes live, helping teams catch problems early and avoid mistakes.

Despite PM's capabilities, the migration to PMWeb still lacks key behavioral modeling features. Without a formula editor, PMWeb is limited in its ability to:

1. **Define business rules through formulas**, such as validating data, adjusting product components based on user input, or calculating premiums dynamically;

2. **Connect formulas to the Business Object Model (BOM)**, allowing them to reference and work with attributes already defined in the product structure;

3. **Make formula creation accessible to business users**, so they can define and manage logic without needing constant support from technical teams.

This gap has significant implications:

- **Fragmented Workflows**: Clients must switch between systems to manage structural definitions (PMWeb) and behavioral logic (PM);
- **Reduced Efficiency**: Dependence on IT for formula creation and validation leads to delays;
- **Limited Flexibility**: The inability to quickly adapt formulas hinders responses to changing market demands or regulatory updates.

The integration of a **formula editor** into PMWeb represents a pivotal step toward resolving these issues. To address the demands of behavioral modeling, the editor must incorporate:

- **Real-time Syntax Validation**: Ensuring that code is correct and aligns with the Business Object Model (BOM) attributes by providing autocomplete suggestions and maintaining validity within a syntactical context;
- **Version Control**: Supporting formula branching and maintaining a historical record of changes;
- **User-Friendly Interface**: Providing autocomplete and contextual suggestions to guide non-technical users;
- **Collaborative Editing**: Allowing teams to work concurrently on formulas, ensuring accuracy and efficiency.

By bridging the gap between PM and PMWeb, the addition of a formula editor will enable insurers to streamline their operations, improve agility, and transition entirely to a web-based environment. This enhancement marks a transformative milestone for PMWeb, positioning it as a comprehensive replacement for its desktop predecessor.

## 1.3 Objective

The objective of this study is to explore the challenges and benefits of *real-time syntax and autocomplete suggestions and validation systems* in *web-based environments* for *data-intensive industries* and to investigate how *data model structures* can be effectively integrated into these systems to complement the syntax of the language with the knowledge of a specific object structure.

The research was conducted to address two primary questions:

1. **What are the primary challenges and benefits of implementing real-time syntax-checking, autocomplete suggestions and validation systems in web-based environments for data-intensive industries?**

2. **How can data model structures be effectively integrated into code editing systems?**

## 1.4 Research methodology

The research methodology aimed at solving the technical as well as the practical challenges of designing an interactive formula editing platform with intuitive integration of the Business Object Model (BOM). The first step involved analyzing existing live code editing platforms with a special focus on issues such as the computational cost of real-time validation, handling big data, and the limitations in collaborative usage. Platforms including CloudCoder, Theia IDE, and VS Code extensions have been investigated, all of which offer differing methods for overcoming these issues.

Second, the study explored how data models could be directly implanted in editing. This involved exploring hierarchical representations, domain-specific languages (DSLs), and semantic annotations that help ensure consistency between user interfaces, business logic, and access rules as the data model evolves. Comparing different platforms, including LSP-based applications and low-code tools, provided insight into their respective limitations, weaknesses, and suitability for real-time environments.

Finally, co-operative system lessons were also taken into account, for instance, in regards to concurrent editing, traceability, and how to maintain consistency within common context. These were then synthesized and interpreted against the research questions, taking into account how pragmatic they are to real-world application in data-intensive sectors such as insurance.

## 1.5 Ethical Considerations

Ethical considerations are fundamental to ensure the integrity, transparency, and societal responsibility of the research and development process. This thesis, focused on the PMWeb application for insurance companies, adheres strictly to established ethical standards throughout its conception, implementation, and documentation.

This work was conducted in full compliance with the *Code of Good Practices and Conduct* of P. PORTO, as demonstrated by the *Declaration of Integrity* included in this document. Furthermore, all principles outlined in the *Order of Engineers' Code of Ethics* and the *IEEE Code of Ethics* were observed, ensuring adherence to professional standards in software engineering and research.

The data used in this project was entirely mock data, created for development and testing purposes. No personal or sensitive information was involved. In terms of software and intellectual property, all frameworks and tools used in this work, including the Language Server Protocol (LSP), Monaco Editor, and Dolt, are either open-source or publicly licensed. Wherever external resources, libraries, or academic contributions were reused, they were properly credited, and in cases where reuse extended beyond citation, permissions were confirmed.

This work also aims to contribute back to the software engineering community by using open technologies and making the main findings accessible to others working on similar problems. Throughout the project, the goal was to maintain a responsible and transparent research process that respects both academic standards and professional expectations.

## 1.6 Document Structure

This document is organized into eight main chapters:

1. **Introduction** outlines what the project aims to achieve, starting with a description of the legacy PM system and the emergence of PMWeb. It defines the research questions and explains the methodology used to address them. Ethical considerations are also discussed to ensure the work was carried out responsibly. Finally, the chapter presents the overall structure of the document and includes a summary of the planning phase.

2. **Literature Review** explores previous studies related to the two research questions. It begins by presenting the questions themselves, followed by the selection of databases and digital libraries used to find the relevant publications. This includes a breakdown of the search terms and Boolean operators applied to retrieve results effectively. The chapter then explains the inclusion and exclusion criteria that guided the selection of articles, and describes the full data collection process, which followed a systematic review methodology. After presenting and discussing the findings in relation to the research questions, the chapter concludes with a synthesis of the key insights gained from the literature.

3. **Technologies** chosen to support the proposed solution. This chapter introduces the Language Server Protocol (LSP) and the Monaco Editor, explaining how they help bring real-time validation and smart editing to web applications. It also presents the idea of version-controlled databases, focusing on Dolt as a Git-inspired tool. These technologies are discussed in terms of how they improve traceability, support teamwork, and help keep data consistent within the PMWeb application.

4. **Analysis:** This chapter presents a functional analysis of PMWeb's enhancement needs, centered on two key requirements: (1) the integration of a formula editor with real-time validation and autocomplete capabilities, and (2) the incorporation of Business Object Model (BOM) metadata into the editing environment. It outlines the rationale behind these needs, the intended improvements in user interaction, and the role of UI/UX considerations in supporting usability and productivity. This analysis sets the foundation for the technical solutions detailed in the following chapter.

5. **Design:** This chapter provides a technical breakdown of the architecture designed to fulfill the functional requirements defined earlier. It explains how the Language Server Protocol (LSP) is integrated into PMWeb's architecture using the C4 model, how version-specific workspaces are created, and how JSON-based BOM structures are converted to Python modules for validation. It also discusses the backend data structure and support of version control, and CI/CD flow.

6. **Implementation:** This chapter translates the design into concrete software components. It describes the development of the custom Node.js-based LSP server, its connection to Pyright, and the integration with Monaco Editor via a reusable frontend component. The chapter also outlines the build and deployment workflows established through Jenkins pipelines and Docker, and how these are used to package and distribute the LSP service in a maintainable and secure way.

7. **Validation:** This chapter presents the methods used to evaluate the implemented solution. The validation process includes both automated and manual testing. Automated tests were used to check the core behaviour of the system, while manual scenarios helped simulate how users would interact with the editor in practice. These tests focused on specific features such as syntax checking, context-aware suggestions, restrictions on what can be edited, and limits on formula length. The aim was to show that the system works reliably, is easy to use, and holds up well under typical usage conditions.

8. **Conclusion:** The final chapter summarizes the contributions of the project, revisiting the initial challenges and how the proposed solution addressed them. It reflects on the system's current capabilities, discusses limitations, and outlines future directions such as dynamic BOM updates, cross-formula imports, and potential support for other languages like OCL. The chapter closes by reinforcing the broader impact of the work on collaborative modeling within PMWeb and similar systems.

9. **References:** Articles referenced throughout the report.
10. **Appendix:** Additional information such as raw data or transcripts.
    - **Appendix A:** Risk Register.

## 1.7 Project Planning

## Main Elements from the Project Charter

This section provides the foundation for the project's scope, objectives, stakeholders, and constraints, ensuring alignment with best practices in project management.

**Main Stakeholders**
The success of the PMWeb enhancement relies on the involvement of diverse stakeholders. These include:

- **Primary Stakeholders**:
    - **Business Users**: Insurance professionals leveraging the system for product configuration;
    - **Software Developers**: Responsible for implementing and maintaining the system;
    - **Company**: Msg Life Iberia gains value with the implementation of this work through various means. For instance, it enables the addition of a new product to the portfolio, supports technological renewal, and adopts a simpler and more commonly used programming language that reduces the learning curve, thereby improving accessibility and efficiency for development teams.
- **Other Stakeholders**:
    - **Academic Community**: Interested in the innovative aspects of formula management and data versioning;
    - **Regulatory Authorities**: Ensuring compliance with industry standards and ethical considerations.

**Benefits**
The project delivers significant technical, business, and academic benefits:

- Empowerment of non-technical users to configure insurance products without IT dependence;

- Enhanced operational efficiency with real-time validation and integrated formula editing;
- Academic contributions to the field of web-based, collaborative formula editing systems;
- Strengthened regulatory compliance and data traceability for insurance companies;
- Cost and resource savings due to optimized workflows and reduced errors.

**Constraints**
The project operates within the following constraints:

- **Technological Dependencies**: Reliance on the chosen frameworks (e.g., Monaco Editor, Dolt);
- **Data Sensitivity**: Restriction on using proprietary or sensitive data during tests;
- **Timeline**: Deliverables must align with pre-defined academic and organizational schedules.

**Assumptions**
Critical assumptions guiding the project include:

- Frameworks and tools will remain relevant and supported throughout the project duration;
- Simulated datasets will effectively replicate real-world conditions;
- The formula editor's design will integrate seamlessly with the Business Object Model (BOM);
- Users will adopt and adapt to the new formula editing system without significant training barriers.

## Scope of Work

The project scope aligns with a structured WBS to ensure seamless execution. Key phases and deliverables are:

1. **PREPD Phase**:
    - 1.1 Deliverable: Formalization
    - 1.2 Deliverable: Dissertation Preparation Document
    - 1.3 Deliverable: Dissertation Preparation Document Presentation

2. **Design Phase**
    - 2.1 Deliverable: Selection and analysis of tools and frameworks.
    - 2.2 Deliverable: Architectural design of the integrated formula editor.
3. **Implementation Phase**:
    - 3.1 Deliverable: Development and integration of the formula editor into PMWeb.
    - 3.2 Deliverable: Bom Integration.
    - 3.3 Deliverable: CI/CD.
4. **Validation Phase**:
    - 4.1 Deliverable: Testing of formula editor functionality and performance.
    - 4.2 Deliverable: User feedback reports.
5. **Documentation and Presentation Phase**:
    - 5.1 Deliverable: Thesis documentation and final presentation.

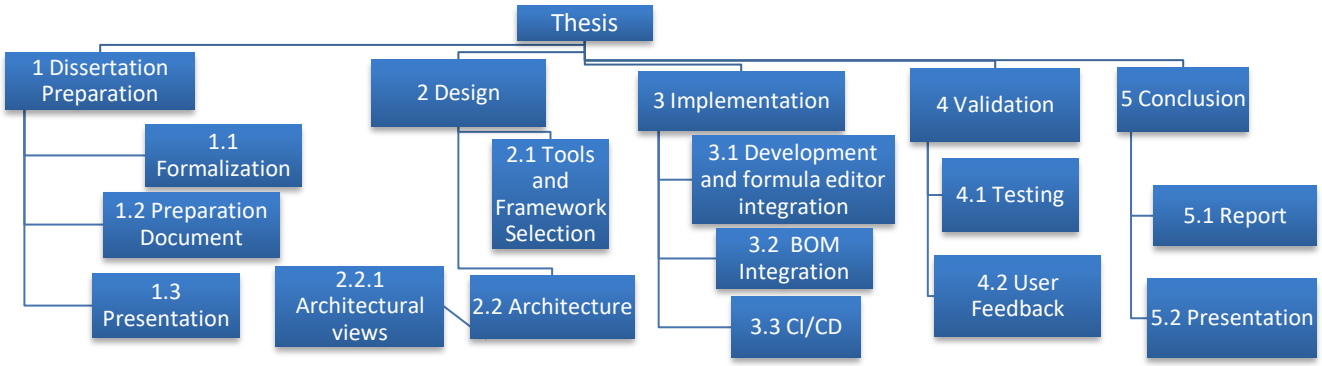


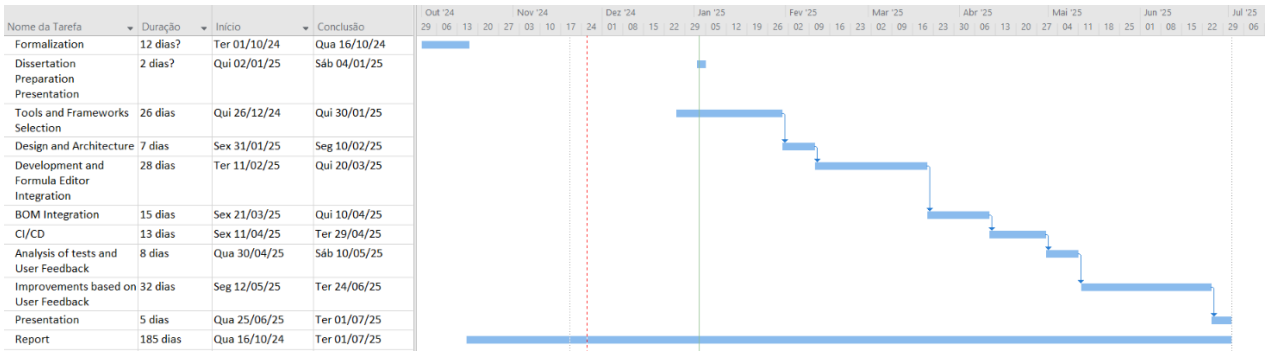

Figure 1 - WBS Project Deliveries

### Integrated Project Schedule

The Gantt chart represented in Figure 2 - Gantt Project based on the Figure 1 - WBS Project Deliveries ensures all tasks are tracked and completed within deadlines. Key milestones include:

- **Formalization**: Oct 1, 2024 - Oct 16, 2024
- **Dissertation Preparation Documentation:** Oct 16, 2024 - Jan 4, 2025
- **Dissertation Preparation Presentation:** Jan 2, 2025 - Jan 4, 2025

- **Tools and Frameworks Selection**: Dec 16, 2024 – Jan 30, 2025
- **Design and Architecture**: Jan 31, 2025 – Feb 10, 2025
- **Development and Formula Editor Integration**: Feb 11, 2025 – Mar 20, 2025
- **BOM Integration**: Mar 21, 2025 – Apr 10, 2025
- **CI/CD**: Apr 11, 2025 – Apr 29, 2025
- **Analysis of tests and User Feedback**: Apr 30, 2025 – May 10, 2025
- **Improvements based on User Feedback**: May 11, 2025 – 24 June, 2025
- **Presentation**: 25 June, 2025 – 30 June, 2025
- **Report**: Feb 2025 – July 2025



| Nome da Tarefa | Duração | Início | Conclusão |
|---|---|---|---|
| Formalization | 12 dias? | Ter 01/10/24 | Qua 16/10/24 |
| Dissertation Preparation Presentation | 2 dias? | Qui 02/01/25 | Sáb 04/01/25 |
| Tools and Frameworks Selection | 26 dias | Qui 26/12/24 | Qui 30/01/25 |
| Design and Architecture | 7 dias | Sex 31/01/25 | Seg 10/02/25 |
| Development and Formula Editor Integration | 28 dias | Ter 11/02/25 | Qui 20/03/25 |
| BOM Integration | 15 dias | Sex 21/03/25 | Qui 10/04/25 |
| CI/CD | 13 dias | Sex 11/04/25 | Ter 29/04/25 |
| Analysis of tests and User Feedback | 8 dias | Qua 30/04/25 | Sáb 10/05/25 |
| Improvements based on User Feedback | 32 dias | Seg 12/05/25 | Ter 24/06/25 |
| Presentation | 5 dias | Qua 25/06/25 | Ter 01/07/25 |
| Report | 185 dias | Qua 16/10/24 | Ter 01/07/25 |

Figure 2 - Gantt Project

## Costs

Potential cost factors:

- Cloud services for testing environments;
- Software licenses;
- Specialized equipment for performance and validation tests;
- Conference or journal submission fees for academic publication.

## Project Risks Identification and Management

The following risks on Table 1 - Project Risk Identification have been identified, along with mitigation strategies:

| Risk Description | Cause | Effect | Mitigation Strategy |
|---|---|---|---|
| **Lack of access to sensitive data** | Regulatory restrictions | Inability to test real-world scenarios | Use high-fidelity synthetic datasets |
| **Technology obsolescence** | Rapidly evolving frameworks and tools | Loss of compatibility | Regularly update dependencies |
| **Integration challenges** | Complexity of BOM and formula editor | Delays in functionality implementation | Allocate buffer time for debugging |
| **Stakeholder feedback delays** | Limited availability of business users | Iterative adjustments delayed | Set fixed timelines for reviews |
| **Performance bottlenecks in real-time validation** | Large datasets and formula complexity | Reduced responsiveness | Optimize with cloud resources |

Table 1 - Project Risk Identification

Appendix A contains the Risk Register table.

# 2 Literature Review

This review of the literature examines prior work on enforcement of code editing and validation systems in web-based environments, particularly in data-intensive sectors, and integration of data model structures that are in synergy with the syntax of the language and knowledge of an existing object structure. The objective of this chapter is to present how these systems address both the technical challenges and benefits of decision-making as well as real-time collaboration, particular in the applications of DSLs within formula editing in finance, healthcare, and logistics.

The two broad research questions guide the literature review, helping to organize the analysis and stay relevant to the study's goals. The review considers the role of DSLs in the improvement of system performance, the application of low-code platforms, and the manner in which collaborative tools can function in cloud-based systems. It also displays the data sources used, as well as the keywords and standards used when choosing studies with relevance. The process followed for collecting and filtering the literature is described, and the findings are organized according to their contribution to each research question.

## 2.1 Research Questions

1. **RQ1:** What are the primary challenges and benefits of implementing real-time syntax-checking, autocomplete suggestions and validation systems in web-based environments for data-intensive industries?

2. **RQ2:** How can data model structures be effectively integrated into code editing systems?

This research began by addressing two key questions. The first (RQ1) looked into the main challenges and possible benefits of building code editing and validation tools for the web. It focused on technical issues like performance, scalability, and keeping everything in sync when

more than one person is involved, as well as the potential to improve productivity and provide users with immediate feedback during modeling.

The second question (RQ2) asked how data model structures could be integrated into code editors in a way that actually improves how users work. The goal was to understand how these models help create systems that are more flexible, easier to update in real time, and capable of handling large sets of data across different scenarios.

## 2.2 Methodology

### Data Sources

Three academic databases were chosen for the literature review because they offered reliable and relevant material for the topic under study. These platforms offer broad access to academic work in software engineering and computer science, particularly in areas such as real-time systems, code editors, and data model integration. The selected sources are shown in the Table 2 - Data Sources.

| Identifier | Database | URL |
|---|---|---|
| **DS1** | Google Scholar | https://scholar.google.com/ |
| **DS2** | ACM Digital Library | https://dl.acm.org/ |
| **DS3** | IEEE Xplore | https://ieeexplore.ieee.org/Xplore/home.jsp |

Table 2 - Data Sources

These databases were chosen for their coverage of peer-reviewed publications, conference proceedings, and technical reports. They provided a reliable foundation for identifying studies directly related to the two research questions.

### Search Terms

In this section, the most relevant keywords for the research questions are identified. The search terms were selected to cover key concepts related to code editing systems and data model integration. These include terms such as **“**code editor**”**, **“**IDE**”**, **“**real-time validation**”**, **“**DSLs**”**, and **“data model”**.

The queries were developed using a combination of Boolean operators (AND, OR) to ensure that the search returns relevant results for both RQ1 and RQ2.

1. **RQ1 Search Query**

("code editor" OR "IDE") AND ("online" OR "web-based") **IEEE**

("code editor" OR "IDE") AND ("online" OR "web-based") AND (“syntax”, “validation”, “autocomplete”) **Google Scholar and ACM**

These queries capture publications related to code editing and validation systems in web-based environments, with a specific focus on syntax validation and autocomplete features, which are key components of real-time formula editing systems. There was a need to use two search queries because IEEE wasn’t returning results with the more complex one.

2. **RQ2 Search Query**

("code editor" OR "DSL") AND ("data model" OR "custom data type") AND ("integration" OR "embedding" OR "accessibility" OR "reuse") **IEEE and ACM**

("code editor") AND ("data model" OR "custom data type") AND ("integration" OR "embedding" OR "accessibility" OR "reuse") **Google Scholar**

This query searches for articles on the integration of data model structures into code editing systems. The terms “data models,” “semantic annotations,” and “DSL” are used to explore how frameworks enable seamless integration of complex data structures across application layers, facilitating dynamic, real-time adaptability and interactivity. There was a need to use two search queries because Google Scholar was returning too many results, related to the usage of the term DSL.

By combining these terms with Boolean operators, the searches are tailored to return publications that address the challenges and benefits of integrating advanced data models into code editors. These queries were executed within the IEEE Xplore, Google Scholar and ACM Digital Library data sources.

An analysis of the results will then be conducted to extract valuable information to answer the research questions.

## Eligibility Criteria

The eligibility criteria for the articles included in this literature review are as follows:

**Inclusion Criteria (IC):**

- **IC1**: The source explores the use of code editing systems or validation systems in web-based environments. This includes studies that focus on code editors (IDEs), syntax validation, and real-time collaboration within the context of data-intensive industries (e.g., finance, healthcare, logistics);
- **IC2**: The source needs to address the integration of data model structures in code editing systems. Specifically, the research should focus on how those models can be used to support dynamic decision-making processes and real-time formula editing.

**Exclusion Criteria (EC):**

- **EC1**: The source does not provide practical scenarios or applications of code editing systems or data model structure integration. Articles that are too theoretical, without real-world implementation or case studies, will be excluded;
- **EC2**: The source is not published in English. Only English-language articles will be considered to ensure accessibility and consistency in reviewing the literature;
- **EC3**: The source was published before 2005.

These eligibility criteria were designed to ensure that the articles included in this review are directly relevant to both code editing systems in web-based environments and the integration of data models in dynamic decision-making systems.

## Data Collection Process

The data collection process for this literature review follows the *PRISMA systematic review process*[1] ,which ensures a structured and transparent approach to gathering and analyzing relevant studies. The process followed can be seen in Figure 3 - PRISMA systematic and consists of three main steps:

**1. Identification:**

The process began with a search across databases like IEEE Xplore, Google Scholar, and the ACM Digital Library, using the previously defined criteria. The goal was to find studies that explore both the challenges and advantages of using code editing systems and domain-specific languages based on data models, especially in web applications that rely on dynamic, data-driven decision-making.

**2. Screening:**

- **Phase 1**: All retrieved articles are categorized as either "*Relevant*" or "*Irrelevant*". In this phase, articles that do not meet the eligibility criteria, such as those that focus on unrelated technologies or lack practical application, are discarded.
- **Phase 2**: In the second phase, the remaining articles are analyzed in greater detail to determine their relevance to the research questions. Articles that do not specifically contribute to the understanding of real-time editing systems or data model integration in dynamic decision-making are excluded.

**3. Inclusion:**

The final step was to filter the results and keep only the studies that directly addressed the research questions. These articles are included in the review, and information is extracted to

address the research questions, including the challenges, benefits, and best practices of integrating code editing systems and data model-based DSLs in data-driven environments.

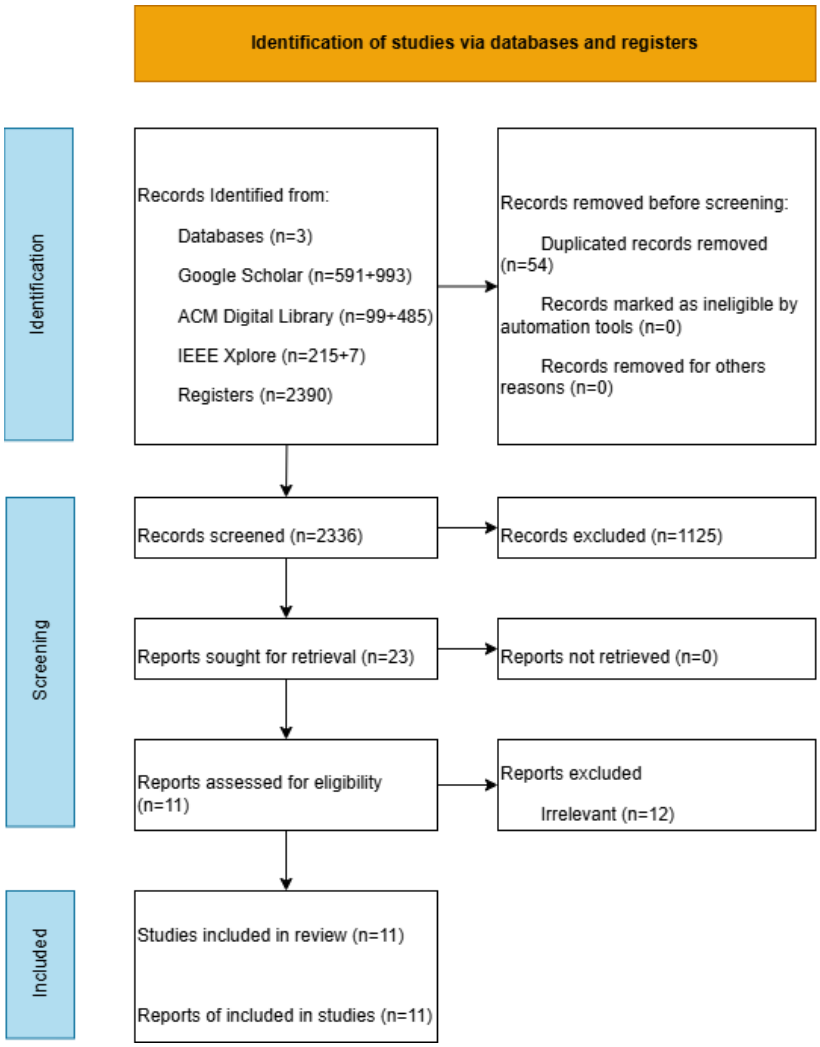



Figure 3 - PRISMA systematic

## Results

The Table 3 - Research Question Results contains all the analyzed studies to answer the research questions.

| Research Questions | Studies |
|---|---|
| **RQ1** | [2],[3],[4],[5][6],[7] |
| **RQ2** | [7],[8],[9],[10],[11],[12] |

Table 3 - Research Question Results

## 2.3 RQ1 - What are the primary challenges and benefits of implementing real-time syntax-checking, autocomplete suggestions and validation systems in web-based environments for data-intensive industries?

Implementing real-time formula editing and validation systems in web-based environments for data-intensive industries presents both significant challenges and notable benefits. In these industries, tools that provide fast feedback and allow quick changes can greatly improve a company's efficiency. Still, building web-based systems that support collaboration is no easy task. It requires managing performance under load, handling multiple users at once, and keeping validation accurate as the system grows.

### Challenges

One of the main challenges is making sure the system stays fast while still catching errors and validating syntax accurately as users' type. Because formulas are often updated continuously as users type or revise them, the system needs to parse and validate input on the fly. When the logic becomes complex or the underlying datasets are large, this can place a noticeable load on performance. In web-based environments, where responsiveness is critical to user experience, managing these tasks efficiently without slowing down the interface becomes a significant technical concern. As highlighted in various studies, this issue is particularly evident in systems that need to support multiple users working on the same document simultaneously, as seen in collaborative environments like Eclipse Theia [2]and VS Code extensions for collaborative modelling [3].

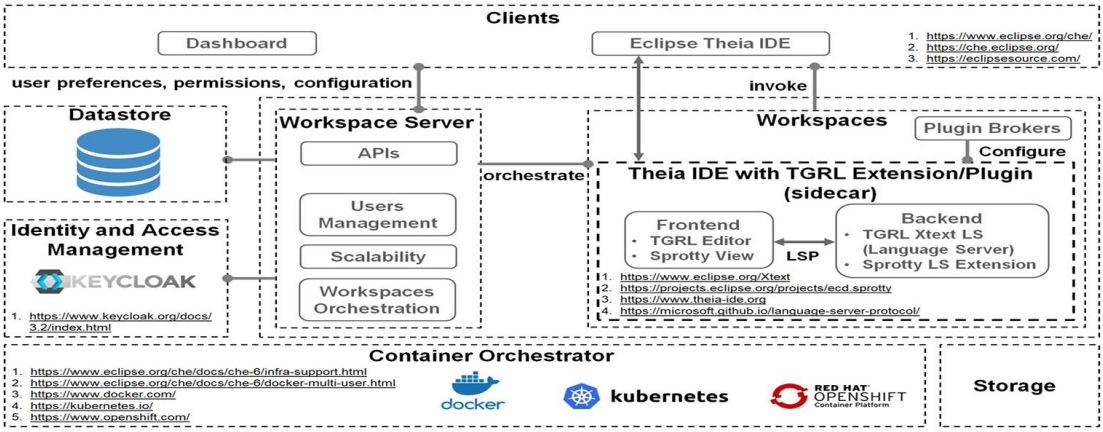



*Figure 4 - tColab Framework for Collaborative Modelling with Theia [2]*

Real-time systems also face performance constraints related to frequent serialization and deserialization of models, which can create delays, especially when the models are large or contain complex interdependencies. Ensuring a responsive experience in such data-intensive environments remains a significant challenge [7].

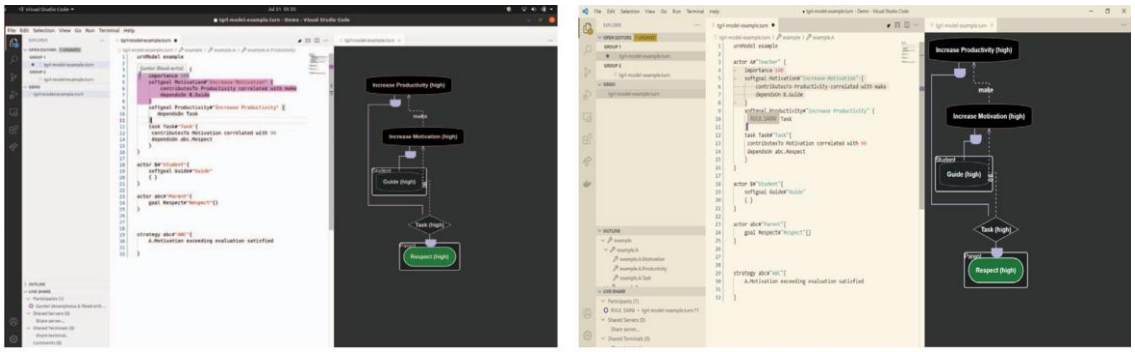
Implementing web-based code editing systems for data-intensive industries requires robust infrastructure. CloudCoder [4], when deployed for multiple classes with over 100 simultaneous users, experienced performance issues on small EC2 instances. A large RAM capacity host was needed to support around 170 students, costing approximately $380 per month. CodeStruct [5] also had previously several unknown bugs that forced participants to restart the editor (on average 1.9 times per participant) during 6 hours of usage. This highlights the potential challenge of maintaining system stability and performance, especially in data-intensive environments where large amounts of code or data might be involved. As mentioned, performance overhead related to frequent validation and data transmission is an ongoing issue in real-time web-based systems [7].

### Scalability and Collaboration Barriers

Scalability is another major concern when building web-based formula editors. Beyond handling syntax validation, these systems must also process large and dynamic datasets that may change as users interact with them. To keep features like autocompletion and live validation responsive, the system needs to be designed with performance and scalability in mind, formula evaluation, fast and responsive, even as the amount of data grows. This is particularly relevant in data-heavy domains, where users may work with extensive tables or interact with predictive models that update in real time. Keeping the system responsive while managing complex references and updating models on the fly becomes increasingly difficult, especially when multiple users are working at the same time [7].

Although these editors support multi-user workflows, editing the same formula at the same time can lead to conflicts. To avoid issues, changes must be coordinated in a way that preserves data integrity and avoids overwriting each other's input. Ensuring consistency across user sessions still requires careful handling of version history and change tracking. Tools that support collaborative editing and multi-view modelling [3] help by letting users work on different parts of the same model while keeping everything aligned across views. Still, adding this kind of functionality to a formula editor demands a robust backend and tight integration with the frontend, both of which must be designed to handle concurrency smoothly. This highlights a challenge with scalability and collaboration, where real-time updates must be effectively synchronized across various users without causing latency issues or data inconsistencies [7].

VS Code extensions enable real-time collaborative modelling, allowing multiple users to edit formulas simultaneously without conflicts, as it's represented in Figure 5 - Two modellers in a collaboration session[3]. By using CRDTs (Conflict-Free Replicated Data Types), users can collaborate editing code in real time, ensuring consistency across edits without conflicts. However, integrating these technologies into a real-time formula editing system requires sophisticated backend infrastructure and seamless integration with frontend interfaces [3]. The need for robust conflict resolution mechanisms and backend support remains a critical issue in such collaborative environments.



(a) Modeller 1 (b) Modeller 2

Figure 5 - Two modellers in a collaboration session[3]

Implementing real-time features using technologies like Google Web Toolkit's RPC can be complex due to the asynchronous nature of web communications. This required creative solutions, such as nested RPC calls and assumptions about maximum file numbers [4]. Real-time validation in such settings, particularly in environments that require high data throughput, necessitates creative approaches to minimize communication delays[7].

## Opportunities and Benefits

Despite the technical effort involved, having real-time formula editing and validation brings clear benefits. Users receive instant feedback while typing, which helps them spot and fix errors early, rather than dealing with them in later phases. Tools already discussed like CloudCoder, Eclipse Theia, and CodeStruct show how effective this can be, catching small issues right away makes the editing process faster and more reliable. In industries that rely on large datasets and fast decisions, this responsiveness allows users to adjust models on the fly and make sure outputs reflect the most up-to-date information. It’s a practical way to reduce the risk of errors in calculations and predictions, especially when decisions have financial or operational consequences [7].

Even without real-time simultaneous editing, formula editors can still support collaboration by enabling teams to contribute to the same models over time. This is especially helpful in projects where data scientists, analysts, and domain experts all need to define or rebuild decision logic. Version control and structured workflows make it easier to review changes, experiment with safety, and align updates around a shared understanding of the data.

It also simplifies version control by keeping track of changes and making it possible to revert when needed, an important feature for maintaining consistency in fast-paced workflows. Tools such as Eclipse Che and Theia IDE [2] offer collaborative editing environments that support multiple users and integrate with version control systems, making them ideal candidates for this kind of collaborative, real-time formula editing. Cloud-based integration further supports the synchronization of changes and ensures that all users are working with the most up-to-date information in real time[7].

One advantage of using web-based systems is that they can connect to cloud services for tasks like validation and data processing. This is especially useful in data-heavy environments, where complex computations can run on remote servers instead of relying on the user's local machine. As a result, the system remains responsive while still handling large-scale modeling work in the background. The integration of cloud infrastructure into these systems also allows for real-time data synchronization, ensuring that all users are working with the most up-to-date data at any given moment. Cloud-based systems like Theia IDE [2] facilitate this integration by providing a modular framework for creating customizable, web-based development environments that can be adapted to different use cases, from formula editing to collaborative modelling.

### Role of the Language Server Protocol (LSP)

In the context of real-time formula editing, Language Server Protocols (LSPs) have emerged as a highly effective solution. LSPs enable rich code editing features such as syntax validation, autocompletion, and error highlighting by decoupling the code editor from the language-specific processing logic [6]. Real-time input validation via the Language Server Protocol (LSP) improves efficiency by identifying errors, suggesting corrections, and auto-formatting documents, significantly reducing time spent on debugging. LSPs support scalability by offloading language-specific tasks to external servers, making them a highly effective choice for real-time formula editing systems in data-intensive environments[7].

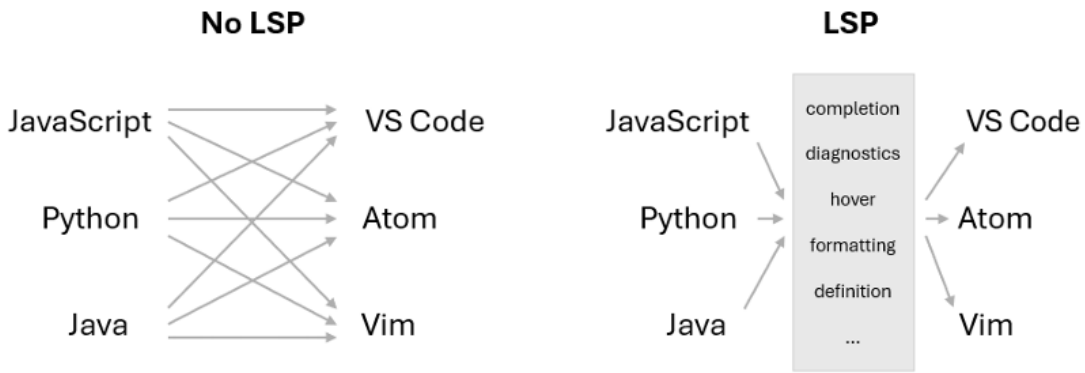



Figure 6 - Visualization of the coupling of language servers and IDEs with and without LSP[6]

This allows formula editing systems to support a wide range of languages and custom DSLs (domain-specific languages) without needing to build language support directly into the editor. LSPs offload the heavy lifting of syntax checking and error detection to a dedicated language server, which can be easily extended to support new languages or custom formulas. This makes them an ideal choice for building scalable, responsive web-based formula editors that can handle complex, data-driven decision models. The modularization of LSPs enhances their adaptability, allowing developers to focus on user experience without worrying about intricate language-specific validation logic[7].

Several studies highlight the value of using the Language Server Protocol (LSP) to support real-time feedback in web-based development environments. LSPs are often paired with cloud backends to handle tasks like syntax validation and code analysis, reducing the load on the client and improving responsiveness. Although most implementations still rely on structured collaboration rather than live multi-user editing, the protocol's ability to keep validation accurate and up to date has made it a popular choice in systems that require both performance and consistent feedback during modeling.

## 2.4 RQ2 - How can data model structures be effectively integrated into code editing systems?

Integrating data model structures into editors is not without its pitfalls. The software must remain stable but impose sufficient flexibility in order to allow for change. Certain frameworks, covered later, show that this type of integration can be done. They offer examples based on real-world usage where editors with complicated data are handled in a productive way.

### Usage of JSON in Domain Modeling

One of the widely used approaches is the use of JSON to represent domain models. Applied in formula editing, JSON patterns can capture objects like Person or Transaction, which programmers can call directly inside the codebase, and the formulas will react to these new objects immediately. For instance, a formula used for calculating an insurance premium may take the Person object as input and adjust the output based on attributes like age or address. Because JSON supports hierarchical nesting, it mirrors the structure of real-world entities, which makes it easier to work with complex decision logic in a flexible and consistent way.

McNutt's survey on JSON-style Domain-Specific Languages (DSLs) for visualization [8] shows that JSON-based grammars can be a practical way to model complex decision systems in a consistent and structured format. This kind of standardization is especially useful in code editors, where users need to define custom objects and work with them directly inside data models. Because JSON supports both simple and deeply nested structures, it works well for expressing decision logic that relies on multiple layers of abstraction and interaction.

### Designing Interactive Models with JSON and ASTs

Web-based graphical model editors are making it easier to integrate JSON-based data structures into code editing workflows. In recent years, many development environments have moved away from heavyweight desktop IDEs toward more lightweight, browser-based tools. These newer systems often represent models using Abstract Syntax Trees (ASTs), which helps structure the data in a way that's easier to process and visualize. These ASTs can serve as representations of complex data models in real-time applications. By adopting Langium, a TypeScript-based framework for model management, and extending it with a model server API,

JSON can be effectively used to represent dynamic, evolving models in a lightweight, efficient format [7]. The AST data structure, a tree-like representation of the model, can be serialized in JSON, allowing web-based editors to dynamically manage and update these models as new data is introduced.

### Graphical Editors and Projectional Interfaces

*Protectional editors,* as discussed by McNutt and Chugh [9], offer a compelling enhancement to this integration by enabling intuitive, visual interaction with JSON data structures. These editors allow users to interact with the underlying JSON objects through a graphical interface, making it easier to visualize and modify complex decision models in real time. For instance, when a user updates the properties of a Person object in the model editor, the linked formula editor can immediately reflect those changes in its calculations. This level of interactivity makes it easier to experiment with different inputs and quickly see how they affect the results, which helps users build and test decision models more effectively.

### Low-Code Platforms and Automated DSL Generation

Automated code generation in web-based process management systems offers useful ideas for improving formula editors and JSON-based DSLs. With the help of flow-based programming and low-code platforms, users can build decision models visually, for example, by using flow diagrams, and have the corresponding JSON-based formulas generated automatically. This approach speeds up development and often results in cleaner, more consistent code. Some tools allow users to design data structures using a UI, where the JSON structures behind it are automatically generated in the background. This makes it easier to run and update formulas in real time within web applications.

Low-code platforms also help bridge the gap between technical and non-technical users. People without a programming background can contribute to building and modifying data models, while still working within a system that enforces structure and consistency. This automation of decision logic, paired with flow-based design tools, supports dynamic real-time decision systems where JSON serves as the backbone for data structuring and model generation. In fact, the approach demonstrated in this study led to 98.68% improvements in development processes and a 36.01% reduction in code size, highlighting the efficiency gains provided by automated code generation in web-based systems[10].

The integration of low-code platforms and automated code generation into JSON-based DSLs will allow users to design decision models more efficiently, making them adaptable to real-time changes in data and improving the overall flexibility and responsiveness of decision-making systems.

### Custom JSON-Based DSLs for Domain Adaptability

JSON-based DSLs make it easier for code editors to handle data that's specific to a given domain. For instance, users can define objects like *Customer*, *Product*, or *Event* that match the structure of their industry. This lets them write formulas that respond to changes in those objects, making it easier to adjust decision rules as business needs change.

Still, there are challenges. Working with large JSON datasets in real time can cause performance issues, especially when updates are frequent or the data volume is high [9]. In those cases, techniques like data compression or switching to binary formats such as Protocol Buffers can help reduce the processing load. In distributed environments, scalability becomes a concern, especially when many users interact with the system at once. Even without real-time co-editing, a formula editor still needs to handle multiple validation requests efficiently and keep performance consistent as usage grows. This requires backend strategies like caching, load distribution, and database tuning to avoid delays and keep the user experience smooth

### Beyond JSON: Declarative Modeling with WebDSL

Moving pass JSON-based solutions, *WebDSL* [11]exemplifies the effective incorporation of data models through its declarative data modeling language, which allows developers to define entities, relationships, and constraints all in one place. For example, a model might include a User with a username property and a list of Post objects. Each Post could be required to include non-empty text. By keeping everything, structure, rules, and relationships within the same language, it becomes easier to manage the logic without switching between different tools or technologies. Keeping all of this within a single system makes it easier to manage and reason about the model as a whole. This cohesive framework reduces the cognitive load on developers and minimizes inconsistencies across components. This solution stages are displayed in Figure 7 - WebDSL compiler stages and compilation process [11].

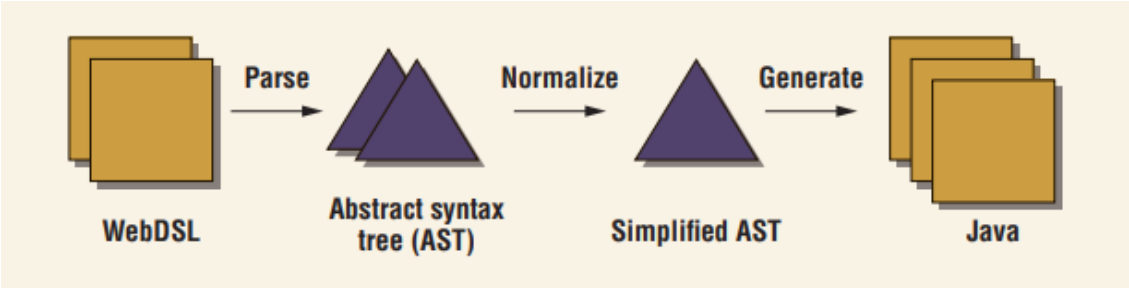



Figure 7 - WebDSL compiler stages and compilation process [11]

A key feature of WebDSL is its linguistic integration, where the data modeling language shares a common type of system and syntax with sublanguages for defining user interfaces, access control, and application logic. This integration ensures that data models are not treated as isolated artifacts but are seamlessly tied to the application's other layers. For example, when a user interface references a data model property, such as displaying a user's username, the

system verifies the property's existence at compile time. Similarly, access control rules defined for entities, such as restricting edits to a post's author, are directly validated against the data model. This cross-layer validation prevents runtime errors and enhances the reliability of the application.

WebDSL also leverages template-driven user interface definitions to bind data models to the visual and interactive elements of the application. For example, an editPost page can bind a Post entity's text property to a text area, enabling bidirectional data updates. In WebDSL, the connection between the interface and the data model is tightly integrated with validation and transaction handling. When a user makes a change through the interface, that change automatically updates the underlying model and respects any constraints that have been defined. This setup allows the code editor to provide real-time feedback while enforcing both syntactic and semantic rules, helping developers catch issues as they work[11].

Access control is also built directly into the data model. Developers can define rules that control what different users are allowed to do, based on the properties of the data itself. For example, it's possible to enforce that only the author of a post can edit it, and these rules aren't just suggestions or comments in the code; they're actually enforced by the system during both development and runtime.

WebDSL is designed to catch mistakes early in the development process. Instead of waiting for errors to show up at runtime, the compiler checks the full data model and how it connects with the rest of the application. This makes it easier to spot inconsistencies before they cause problems and helps developers work more confidently with complex models. This proactive validation aligns with the needs of modern code editors, where real-time feedback and error detection significantly improve developer productivity and reduce debugging time[11].

### Model-Driven Dialog Design with Dialog DSL

*Dialog DSL* [12] introduces a model-driven approach where data model structures are abstracted into Domain-Specific Models (DSMs). These models define the schema for designing advanced dialogs while abstracting the underlying implementation. When the system is designed around a high-level data model, it becomes easier for non-technical team members to collaborate with developers. For instance, an XML schema can be used to automatically generate parts of the user interface, such as dialog components and interaction flows. The modularity of the approach ensures that complex data models can be broken into semantically cohesive units, referred to as dialog partitions, which streamline the interaction design process and reduce cognitive overload for developers and users alike.

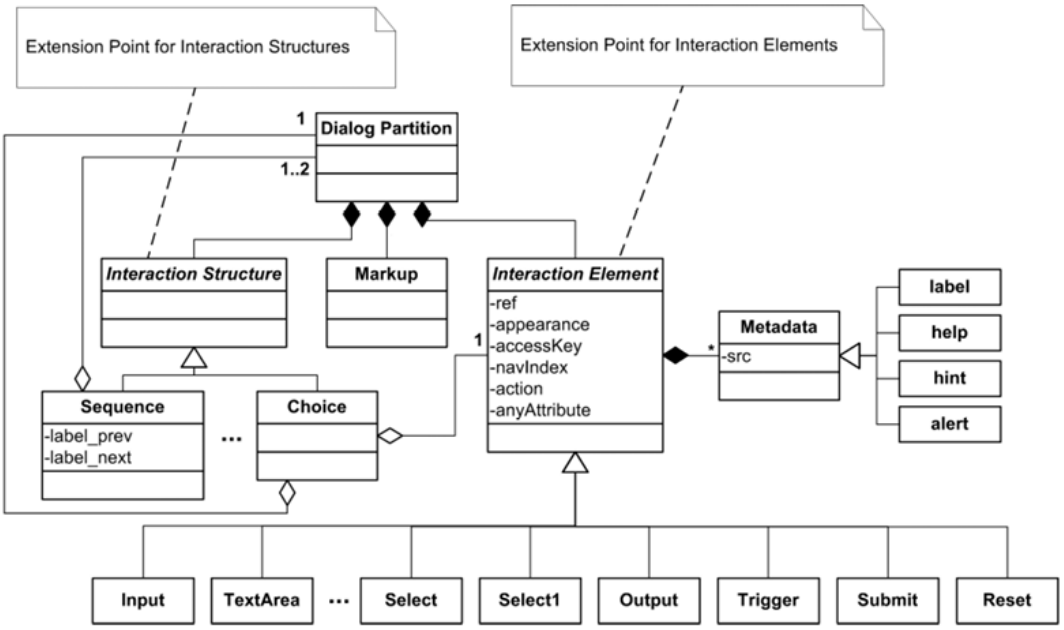



Figure 8 - Simplified excerpt from the DSL's DSM [12]

A two-tiered modeling approach is employed to integrate data model structures effectively. On the first tier, the Petri net-based modeling notation represents dynamic behaviors and transitions within the dialog structure. The Dialog DSL uses visual elements to represent user interactions. Dialog partitions are mapped as places in a Petri net, while transitions show how users move between them. This allows designers to understand how data flows through the interface. Programmers can then map data model elements to UI elements and control layout separate from structure, giving them more control over the final product. One of the benefits of this DSL is that it can be adapted at runtime. Using its SBB framework, it can be supported for different screen sizes or devices without recompiling the application. Models are translated into executable markups like XForms, hence they can be used directly in the system[12].

It also encompasses simple interaction patterns, like "Sequence" and "Choice", that can help to speed up development. A built-in graphical editor supports drag-and-drop assignment of data elements to make it easier to iterate and collaborate during design time.

When changes are made to the data model, only the parts of the dialog that depend on those changes are regenerated. This approach keeps the rest of the structure intact, which is especially useful in fast-moving environments where data requirements change often[12].

## 2.5 Discussion

The research shows that building real-time formula editing systems and domain-aware code editors for the web involves a number of technical challenges. Supporting edits from multiple users can be tricky, especially when dealing with large datasets and complex validations. This

kind of issue is quite common in areas like insurance, where consistency is essential. Even so, the value of real-time editing systems is clear. Rapid syntax verification and helpful tips simplify the development process and make it more reliable. With techniques like the Language Server Protocol, it is possible to separate language reasoning from the visual editor in order to preserve scalability and responsiveness of the system.

Another feature is how data models are integrated into the editor. When one utilizes formats such as JSON or DSLs, the editor can comprehend business logic and data as components of the same framework. Not only does this enhance validation, but it also simplifies the interface to a point where users without technical know-how can work with it easily.

Tools such as WebDSL, graph model editors, and low-code platforms, that we discussed earlier, show you how it is being achieved in practice. They bring together data, logic, and interface within a single platform, reducing context switching among disparate tools and allowing teams to work more productively, particularly in creating systems aimed at specific business needs.

Overall, these insights shaped the direction of the project. They helped define both the requirements and architecture of the final solution, confirming the need for a modular, LSP-based environment that includes structured BOM metadata. The next chapter builds directly on these insights, identifying the technologies that shaped the design of the formula editor and its integration into PMWeb.

# 3 Technologies

The technologies introduced in this chapter have been identified as strong candidates to address the challenges highlighted in the literature review. The review emphasized the need for scalable, robust, and collaborative tools to meet the demands of real-time editing, validation, and efficient workflow management. Several technologies came up in research. The Language Server Protocol (LSP) was a solid choice for introducing syntax checking and autocompletion in real-time. Monaco Editor plays nicely with it, offering a browser-based API that is easy to extend and easy. For working with versioned data in a collaborative setting, Dolt uses a Git-style approach to databases, where modifications can be traced and workflows handled more effectively. This chapter discusses at length how each of these tools works and why they were chosen to be used to help the solution developed in this project.

## 3.1 Key Concepts in Language Server Protocol (LSP) and Monaco Editor

**Language Server Protocol (LSP)**

The Language Server Protocol (LSP) was developed to provide a standardized framework that enables consistent language support across different code editors. Introduced by Microsoft, LSP separates the core language features such as autocompletion, syntax checking, and code navigation from the editor itself, allowing a “language server” to be reused across multiple editors. This architecture simplifies the integration of new languages, as developers can build or adapt a language server once and integrate it with various client editors, thus enhancing scalability and consistency in language support[13].

Historically, one of the primary goals of LSP was to address limitations in traditional Integrated Development Environments (IDEs) by enabling modular language processing capabilities. This modular approach is particularly beneficial in web-based environments where editors must support a range of languages without sacrificing performance [14]. The adaptability and flexibility of LSP have driven its widespread adoption, not only in desktop IDEs but also in cloud-based and embedded web applications, where editors like Monaco leverage it to offer a robust and interactive coding experience [15].

The LSP functions by enabling communication between a client (the code editor) and a server (the language server) through a standardized set of JSON-RPC-based protocols. As explained in Figure 9 - LSP approach to language support. Borrowed from [16] the client sends requests for features such as autocompletion, documentation lookup, or error checking, and the server responds with the relevant data. This decouples the responsibilities of providing language features from the client's UI, allowing for better scalability across multiple environments and reducing the complexity involved in integrating new language support [17].

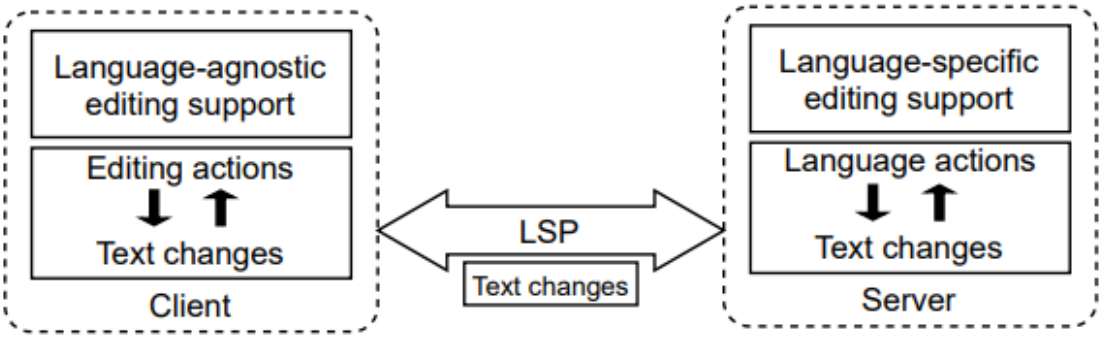



Figure 9 - LSP approach to language support. Borrowed from [16]

LSP's modular nature has extended its use to domain-specific languages (DSLs), allowing their integration into multiple editors without requiring extensive rework for each environment. This decoupling allows developers to maintain and extend DSL support efficiently, especially in complex software projects [18]. Furthermore, the growing adoption of LSP has led to the development of custom servers that cater to niche languages or specific software development needs, such as the implementation of LSP in various formal methods and specification tools [19].

Several case studies highlight the benefits of adopting LSP in real-world applications. For instance, the use of LSP has been instrumental in enhancing tools for formal verification and modelling languages, where the language servers provide robust, language-specific features while maintaining seamless interaction with diverse editing environments [20]. This reflects the continued evolution of LSP in both traditional and emerging fields, further solidifying its role in improving the efficiency of software development processes [15].

## Monaco Editor

The Monaco Editor is a powerful, lightweight code editor that powers the online version of Visual Studio Code. It was developed by Microsoft to provide a rich code editing experience in web-based environments, with features such as syntax highlighting, autocompletion, error checking, and code navigation. Monaco Editor is designed to run in browsers and is optimized for embedding into web applications. Unlike traditional text editors, Monaco can handle a range of programming languages and integrates well with language servers via the Language Server Protocol (LSP), which allows for code validation and smart code completion [13].

Monaco Editor leverages LSP to provide robust support for languages like JavaScript, TypeScript, Python, and others. By connecting to a language server, Monaco can offer advanced features like inline documentation, error highlighting, and code completion tailored to the specific programming language. This interaction between Monaco Editor and LSP explained within Figure 10 - How the Monaco Editor and LSP communicate[21] enables the creation of highly interactive and intelligent coding environments, where developers can write, edit, and test their code with minimal friction[14]. This connection also ensures that Monaco can be adapted to a variety of domain-specific languages (DSLs) and custom configurations, as LSP decouples the language processing from the client editor, allowing Monaco to focus on providing an optimized UI and user experience [15].

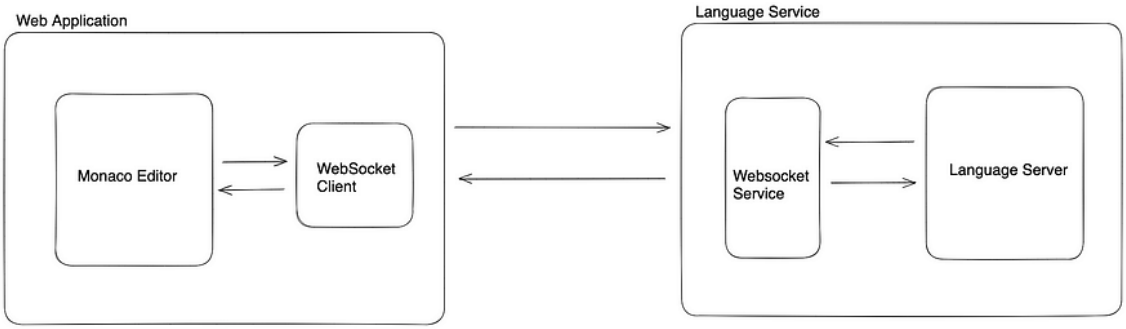



Figure 10 - How the Monaco Editor and LSP communicate[21]

One of the key advantages of Monaco Editor is its flexibility and extensibility. Monaco Editor is modular and easy to extend, which makes it a good fit for both simple web tools and more complex setups [17]. Developers can add things like syntax highlighting, code suggestions, or custom themes without slowing the system down. It's already used in tools like GitHub Codespaces and Visual Studio Online, where people can edit code directly in the browser. Monaco also works well with version control systems like Git, making it easier for teams to collaborate on shared projects[16].

The Monaco Editor isn't limited to standard code editors. It's also used in applications that need rich text editing, like IDEs for educational purposes, online compilers, and platforms for web-based application development. Its strong performance, along with support for plugins and language servers, make sure that it remains a versatile choice for a wide range of use cases,

from simple text editing to more complex scenarios requiring deep integration with other services [18].

### JSON-RPC

JSON-RPC is a lightweight, stateless remote procedure call (RPC) protocol that is normally used for communication between Monaco Editor and Language Server Protocol (LSP) servers. The Language Server Protocol (LSP) connects Monaco to language-specific servers, handling things like syntax checks and autocompletion in the background. This lets Monaco offer real-time feedback without affecting performance, even in browser-based applications.

Monaco Editor uses JSON-RPC to send specific requests to the LSP server, which processes them and returns appropriate responses. The types of requests sent from Monaco to the LSP server depend on the functionalities being used by the editor. These requests typically include:

- **Text Document Synchronization**: Monaco Editor sends updates to the LSP server whenever the user edits code. These updates are captured and sent as a *textDocument/didChange* notification, notifying the server of changes made to the content of the file. This helps the server keep track of code modifications and provides the server with the most recent version of the code for validation and analysis;
- **Autocompletion**: When the user types in Monaco, the editor may send a *textDocument/completion* request to the LSP server, asking for code suggestions or completions based on the current cursor position. The server responds with a list of completions (such as function names, variables, or keywords) that the editor can display for the user to select from. This interaction facilitates code completion as the user writes;
- **Syntax and Semantic Error Checking**: Monaco sends *textDocument/publishDiagnostics* requests to the LSP server, requesting error diagnostics for the current document. The LSP server processes the document and returns a list of errors, warnings, or other diagnostics, which Monaco displays to the user. This ensures that users are immediately notified of issues within the code as they work;
- **Hover Information**: Monaco may send a *textDocument/hover* request when the user hovers over a code element, such as a function or variable. The LSP server processes this request and provides detailed information, such as type annotations or documentation for that element. This feature allows Monaco to provide tooltips and detailed insights for better code navigation and understanding;
- **Go to Definition and Find References**: Another key functionality is the ability to navigate to the definition of a function or variable. Monaco can send a *textDocument/definition* request to the LSP server, which responds with the location of the definition. Similarly, the *textDocument/references* request allows Monaco to retrieve all occurrences of a symbol in the code, enabling users to find and navigate between references;

- **Signature Help**: As the user types a function or method call, Monaco may issue a *textDocument/signatureHelp* request to the LSP server. The server responds with a list of function signatures relevant to the context, highlighting the currently active parameter. This assists users by showing expected parameters and documentation in real-time, improving the accuracy and speed of writing function calls;
- **Document Symbols**: To support code navigation and structure awareness, Monaco can send a *textDocument/documentSymbol* request to the LSP server. The server responds with a list of all symbols (such as classes, functions, variables) defined in the current file. These symbols can be organized hierarchically and are used to populate outline views, breadcrumbs, and "Go to Symbol in File" features, enabling users to quickly understand and navigate the structure of their code.

These requests, among others, are typically sent in JSON format as part of the JSON-RPC protocol. Monaco Editor, through JSON-RPC, can make asynchronous requests to the LSP server for each of these operations, ensuring that the editor remains responsive while performing complex operations like syntax analysis, autocompletion, and error detection.

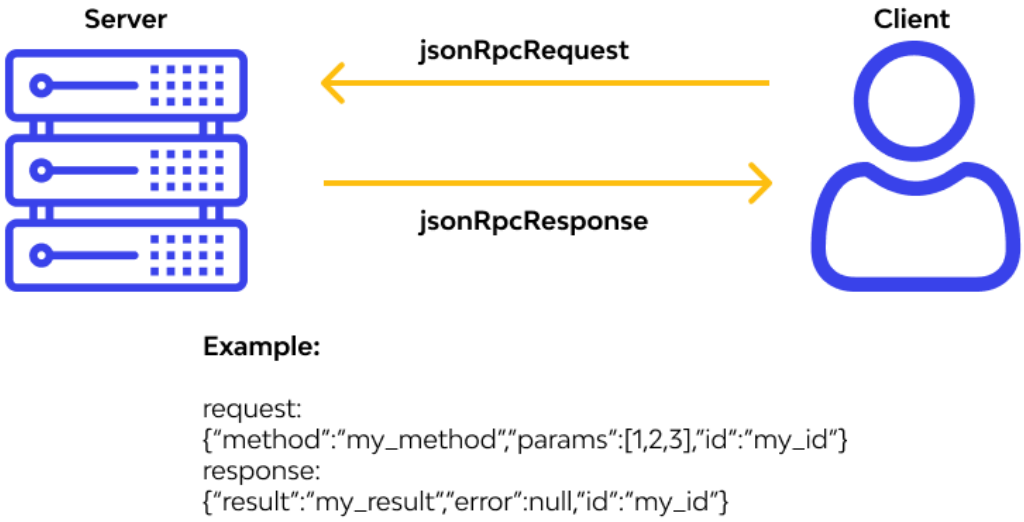



Figure 11 - JSON-RPC example and showcase [22]

Through these interactions such as the ones in Figure 11 - JSON-RPC example and showcase [22], JSON-RPC serves as the transport layer that enables Monaco to deliver a rich, interactive coding experience by allowing real-time language features and validations. The asynchronous nature of JSON-RPC ensures that Monaco can continue to operate smoothly without blocking the user interface, even as it interacts with the LSP server for complex tasks.

The combination of LSP's standardized interface and JSON-RPC's lightweight, stateless communication enables Monaco Editor to support a wide range of programming languages and tools, without tightly coupling the editor to specific language implementations. The protocol allows for scalability, making it easy to integrate additional language servers or extend Monaco's functionality in the future without disrupting existing workflows [16].

## Selection of a Language Server Protocol (LSP)

Integrating an LSP with Monaco adds useful features like real-time validation, autocompletion, and static analysis, making it easier for users to catch mistakes and work more efficiently. Rather than building a new LSP from scratch, we focused on adopting an existing, efficient, and standards-compliant solution that could integrate well into our web-based architecture.

**Why pyright?**
*Pyright*, developed by Microsoft, was selected as the most suitable choice. It is a fast static type checker for Python, built in *TypeScript*, which makes it highly compatible with JavaScript-based environments like PMWeb's frontend stack. Pyright was originally designed for use with Visual Studio Code but can be decoupled and run independently, including within web environments using JSON-RPC over WebSockets [23]. Its integration in the *Pyright Playground* project demonstrates exactly that a live, browser-based Monaco Editor connected to a Pyright server, closely mirroring our intended architecture [24].

Beyond its flexibility, Pyright is notable for its performance and precision. In a comparative case study of Python static type checkers, Pyright not only detected the highest number of real bugs (after type annotations were introduced) but also did so with the best execution time, averaging just 1.83 seconds per bug, compared to 3.13s for MyPy and over 120s for PyType [25]. These empirical results support its use in scenarios that demand both real-time responsiveness and type safety.

Pyright was chosen for its strong handling of Python's type system, especially support for gradual typing through PEP 484 [26]. It lets developers add type hints over time, improving validation without needing to rewrite everything at once.

We also looked at python-lsp-server (pylsp), which supports plugins and works with tools like flake8 and mypy. While it's useful for code quality, its Python-based design made it less suitable for our setup. PMWeb's architecture is heavily Node.js- and Java-based and running a Python service alongside these would introduce operational complexity and increase maintenance overhead. Developers of the company also lack experience with Python. Moreover, although pylsp offers extensibility, it lags behind Pyright in terms of performance and static analysis precision, as shown in controlled experiments [25].

Other alternatives were reviewed as well:

- *Pylance*, built on Pyright, provides rich features but is tightly integrated with Visual Studio Code and not intended for general-purpose use outside of that context [23];

- *Jedi Language Server* offers basic features like autocomplete but lacks comprehensive static type checking;
- *Custom implementations* using frameworks such as *PyGLS* would offer full control but require significant development and maintenance effort [27].

Finally, Pyright has proven value in production environments. At CERN, it was successfully integrated into the CI pipeline of the Rucio data management system, where it helped reduce over 3,000 legacy errors by tracking only new type issues introduced in each commit. Developers used Pyright's precise error messages and integration with GitHub to improve quality assurance without compromising development speed [28].

For all these reasons, *efficiency, web compatibility, type system rigor, and real-world validation*, Pyright was chosen as the LSP to power PMWeb's formula editing capabilities. Its lightweight nature and TypeScript foundation make it particularly well suited to integrate with the Monaco Editor and scale effectively within modern, web-based enterprise platforms.

## 3.2 Database with Data Versioning

Versioning data is important in modern database systems, especially where lots of changes occur on a continuous basis and history has to be traced back. Most standard databases are not as good in this area, so it becomes harder to manage revisions, view earlier versions, or reverse changes when needed. That is where version-controlled databases come in, using ideas from version control tools such as Git, such as branches, commits, and snapshots, to manage data through time.

Versioning is especially useful in projects where many users are editing the same data at the same time. It helps track what changed, when it changed, and who made the change. It also lets users roll back to earlier states if needed. Unlike traditional systems that rely on external tools for tracking, databases like Dolt have these features built in from the start. This makes it easier to manage changes without adding complexity to the application.

### Dolt: A Database for the Git Era

Dolt is a modern database that incorporates Git-like features to manage data versions, making it a standout tool for users needing sophisticated version control over their data. Dolt is an open-source database that brings version control features, similar to Git, into a traditional relational database system. Dolt lets users branch, merge, and create snapshots directly in the database. Since these features are built in, it's easier to keep track of changes as data evolves.

One of Dolt's most powerful features is how it handles collaboration. People can work on various branches and merge later, which is especially useful in projects with ongoing updates or collaboration, like research or financial modeling. It also enables viewing past versions and rolling back changes when needed. In essence, Dolt revolutionizes the way databases handle

versioning by taking inspiration from Git, allowing users to treat their data in a more flexible, trackable, and collaborative manner.

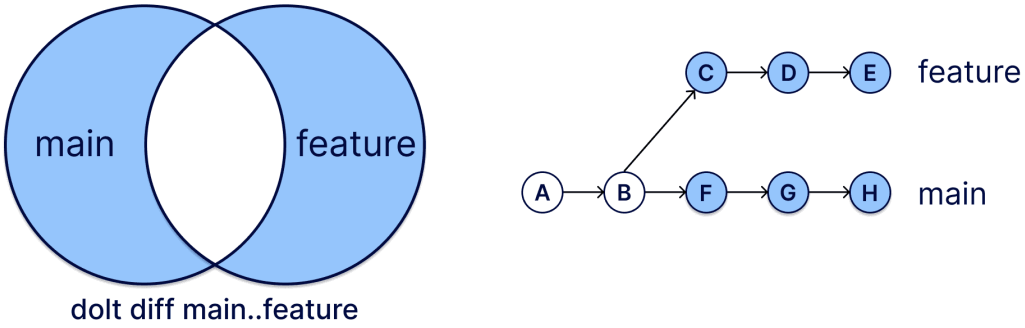



Figure 12 - Dolt Branching [29]

### Real-World Applications and Use Cases

The application of version-controlled databases can be observed across various industries. For instance, in collaborative data science projects, teams often need to explore different preprocessing methods or machine learning models without interfering with each other's work.

Version-controlled systems like Dolt let each team member work on their own branch and combine their changes when ready. QubiCSV, a tool used in quantum computing research. According to Brahmbhatt et al. (2024), it applies versioning to manage complex datasets during qubit control experiments, allowing teams to collaborate and track every change, much like Git does for code [30].

Stenbacka (2024) also points out that features like multiversion indexing and branching make it easier to work with large datasets. Bringing Git-style functionality into databases offers a practical way to track how data evolves. It helps teams collaborate more effectively and reduces the risk of inconsistencies, especially in fast-changing environments [31].

Dolt is a strong example of this shift toward version-controlled database systems. Dolt brings Git-like features such as branching, merging, and snapshots to facilitate easier change management for teams. That allows for easier tracking of progress, collaboration, and auditing of past modifications. It's especially useful in environments where precision and traceability matter, like research, finance, or insurance.

### PMWeb's Use of Dolt for Data Versioning

PMWeb uses Dolt because it supports version control at the database level, something that’s crucial when data changes often and needs to be tracked. Features like branching, merging, and

snapshots help keep a clear record of updates and make it easier to understand how and when changes happened.

One of the main benefits is how Dolt handles team collaboration. Users can work on separate branches and merge their updates later, which is helpful in large projects where different teams manage different parts of the data. Dolt keeps the structure consistent even as multiple people contribute.

PMWeb also uses snapshots to save the state of the data at key points. This is useful for backup, recovery, or simply reviewing how things have changed over time, especially in areas like construction or insurance, where accurate records may be legally required.

By building on Dolt, PMWeb gives its users a way to work together without losing track of what's changed. Even if each user works on just one branch at a time, those branches can evolve in parallel and be merged when ready. This makes versioned data management simpler and more reliable.

### Data Versioning, Branches, and Workspaces

PMWeb organizes data using *workspaces*, which act as primary branches, each supporting its own set of DVs. For example, a workspace like "Development" may contain multiple DVs that evolve as updates are made to insurance data. Users can approve specific DVs within this workspace, effectively "committing" these changes, which are then merged into the main branch. Additionally, PMWeb enables the migration of DVs between branches, allowing, for example, approved DVs in the "Development" workspace to be moved independently to "Production" or as a grouped update. Branching makes it easier to test changes before applying them to live data. In PMWeb, this setup is especially useful for handling insurance data, where updates need to be reviewed carefully before going into production.

By organizing data changes into distinct versions and branching workspaces, PMWeb provides insurance companies with granular control over updates, enhancing reliability and consistency in data management. This makes the dolt database a great choice for this project.

# 4 Analysis

To address the limitations identified in PMWeb, particularly the absence of behavioral modeling functionalities, this chapter presents a functional analysis of the proposed solution. The goal is to outline the expected capabilities from a user's perspective, focusing on the key requirements that must be met to enable efficient, scalable, and collaborative formula modeling in a web environment.

The proposed solution is structured around two main requirements.

## 4.1 Requirement 1: Formula Editor with Real-Time Validation

A central feature required by PMWeb users is the ability to define and edit formulas that describe business logic, such as validations, pricing rules, and conditional configurations. This functionality must be provided in a web-based environment and offer:

- Real-time syntax validation to detect errors as users type;
- Autocompletion to assist with formula construction;
- Support for referencing existing formulas and domain objects;
- A user-friendly interface accessible to both technical and non-technical users.

This editor is expected to reduce dependency on the legacy PM desktop system and empower users to manage formulas autonomously, with immediate feedback and greater flexibility.

## 4.2 Requirement 2: Business Object Model (BOM) Integration

The second core requirement is the integration of the Business Object Model (BOM) into the formula editing context. Formulas must operate on structured domain entities, such as Person or Address, and reference their attributes in a consistent and semantically correct way.

To support this, the system must provide:

- A consistent way to describe and manage BOM structures;
- The ability for the formula editor to recognize and validate formulas within the context of a specific BOM.

This integration ensures that business logic remains aligned with product data definitions, reducing the risk of errors and promoting maintainability.

## 4.3 Other Use Cases and Functional Requirements

Beyond the two primary requirements, the solution should also support:

- Version control for formulas, enabling teams to work on different branches and track changes;
- Formula metadata management, including name, description, context, and return type;
- Search, filtering, and management of formula collections.

While these features are important for a complete user experience, **the focus of this work will be on Requirements 1 and 2**, which are the foundation for enabling behavioral modeling in PMWeb.

## 4.4 UI/UX Considerations

The design process was conducted using *Figma*, where multiple iterations were explored to strike the right balance between usability and feature coverage. In shaping the page layout, interaction flows, and component selection, the team followed *PatternFly version 4*, a widely adopted design system for enterprise web applications. This ensured visual consistency and adherence to accessibility and usability best practices.

The result is a modular, responsive formula editor interface that supports the following primary user flows:

- *Editing the formula expression*: An editor embedded in the page allows users to write or modify the formula logic. Real-time validation feedback is shown directly within the editor(Requirements 1 and 2);

- *Creating a new formula*: Users can define the formula's name, description, context, return type, and parameters;
- *Editing or deleting existing formulas*: All formula metadata and its expression can be modified or removed;
- *Filtering and browsing formulas*: A searchable and filterable list of all existing formulas allows users to quickly locate relevant entries.

These user flows were defined through concrete *use cases*, and the interface was designed to minimize cognitive load while offering maximum control over the modeling logic.

Keep in mind that the design that was defined by the UI/UX expert can suffer changes with the development or project needs, and end up with some differences but the main idea of what's expected here is expressed in the Figure 13 - Presents the main interface layout for formula management and Figure 14 - Demonstrates the modal used to define formula metadata when creating a new formula. Some of the features there will also be developed in the future and aren't part of this project.

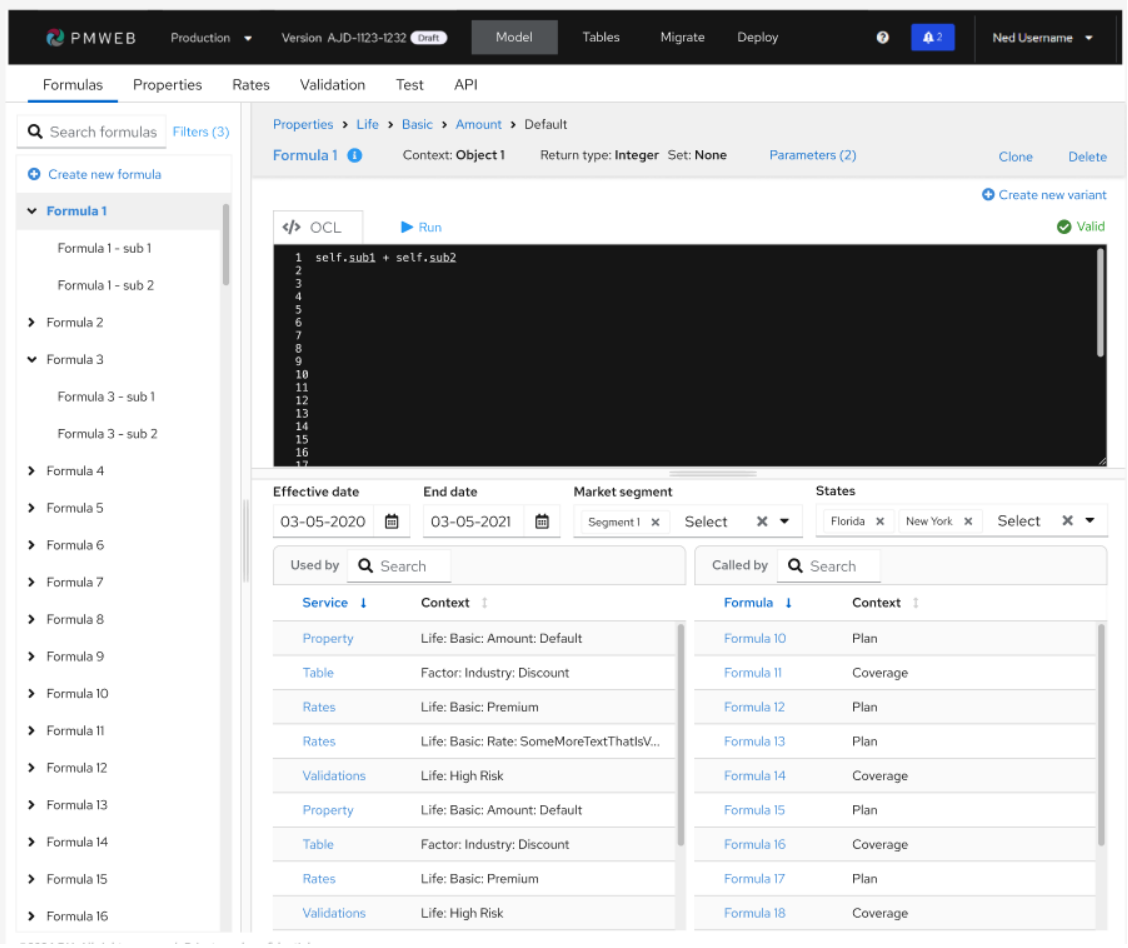



Figure 13 - Presents the main interface layout for formula management

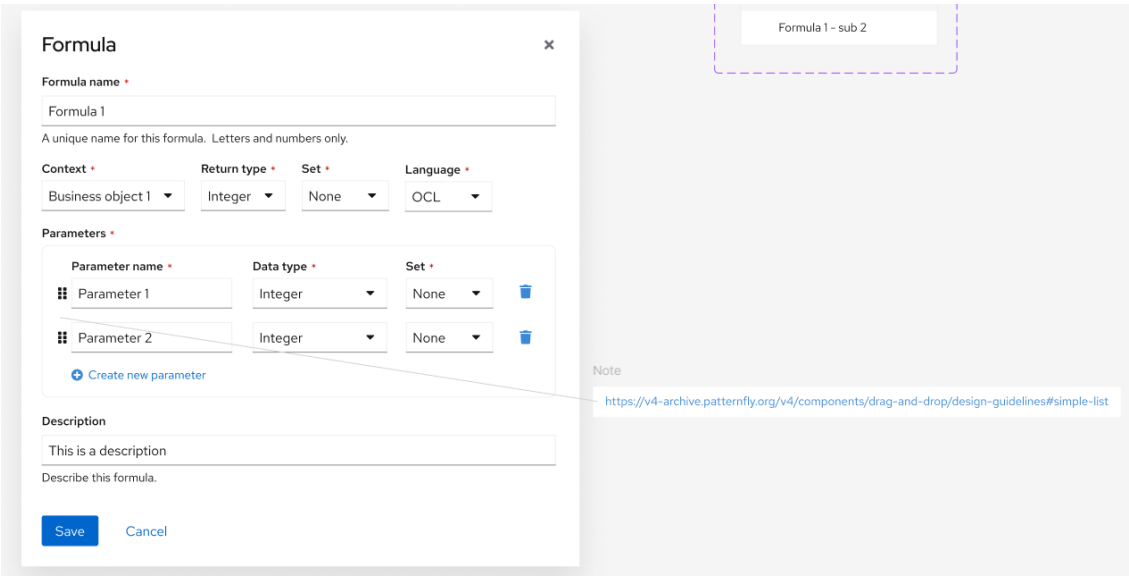



Figure 14 - Demonstrates the modal used to define formula metadata when creating a new formula

This UI/UX approach not only aligns with the underlying system architecture but also empowers users to model business logic with clarity, precision, and confidence. As the formula feature evolves to support multi-language support, multiple variations, and dependency tracking, the interface is designed to grow in parallel, ensuring a future-proof and user-centered experience.

# 5 Design

This chapter presents the architectural design that supports the functional requirements defined in the previous section. The solution addresses two main goals: enabling formula editing with real-time validation (Requirement 1) and integrating the Business Object Model (BOM) into the editing environment (Requirement 2).

To support real-time editing, a new architectural component is introduced, *a Language Server Protocol (LSP)* service. This component is responsible for providing syntax validation, autocomplete suggestions, and static analysis during formula creation. The LSP is accessed via *Monaco Editor*, embedded in PMWeb's frontend, and communicates over *JSON-RPC* using *WebSockets*. This combination offers a familiar, responsive, and extensible experience for both technical and non-technical users.

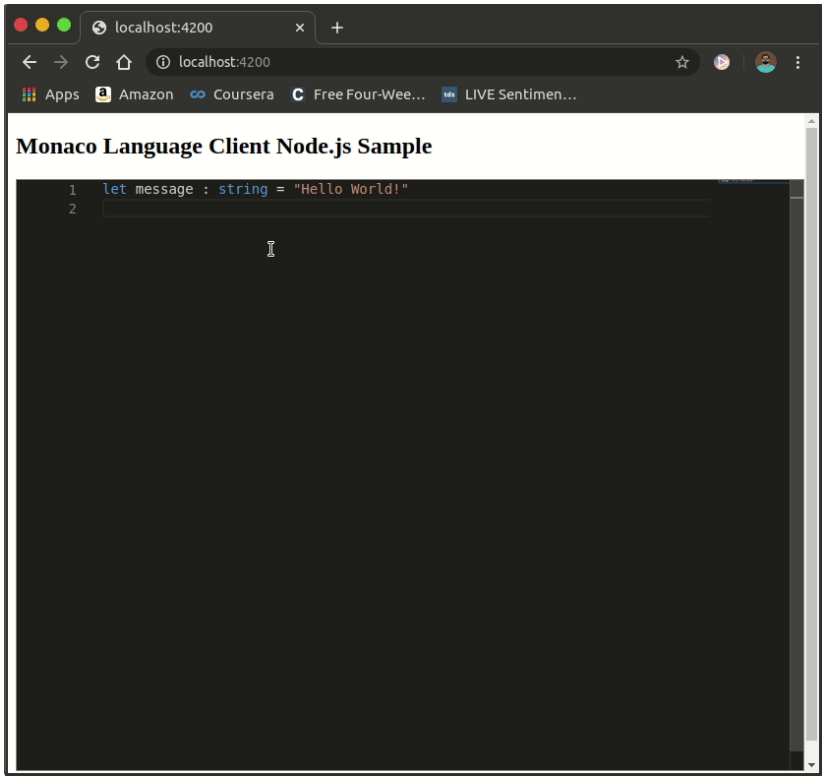



Figure 15 - Monaco editor connection with LSP via JSON-RPC[32]

The second key design requirement involves *BOM integration*. Since formulas operate on structured domain entities, the system must understand and validate formulas within a specific object context. To achieve this, PMWeb supports the injection of a client-specific BOM described in *JSON*, which is transformed into a Python-compatible format for validation by the LSP. This ensures semantic correctness, autocompletion of relevant fields, and alignment with the product's data structure.

To manage collaboration, the system uses Dolt's versioning so teams can work on separate branches of formulas and BOM definitions. This setup makes it easier to test changes without conflicts and keep a clear record of what was changed, important in industries where traceability matters.

The following sections detail how each requirement is addressed from an architectural perspective:

- **Requirement 1**: Describes the design and integration of the LSP service and how it supports the formula editor using Monaco;
- **Requirement 2**: Explains the full flow of BOM integration, including how JSON structures are transformed, injected, and validated during editing;
- Additional sections cover delivery pipelines and version control strategies to ensure the system remains stable, maintainable, and scalable.

Together, these components form the foundation for a robust and extensible formula editing environment within PMWeb.

## 5.1 Requirement 1: Formula Editor with Real-Time Validation

This new component, the Language Server Protocol, was introduced as the architectural response to the first identified requirement: the need for a formula editor capable of real-time validation, intelligent suggestions, and immediate feedback. Its integration enables the system to support behavioral modeling in a web-based environment without relying on external tools. In this section we will display how the integration of this component was designed to fit into PMWeb's architecture

To support a clear and consistent design process, the C4 model will be applied, focusing primarily on Level 1 (Context) and Level 2 (Container). This helps visualize the main components of the system, how they interact, and where the new LSP will fit within the overall architecture of PMWeb. Using this model makes it easier to ensure that the integration of the LSP complements PMWeb's structure without disrupting its modular design or existing workflows.

On the front end, the Monaco Editor will act as the user interface for writing and editing formulas. To support real-time validation and feedback, the system will rely on an LSP. Rather than developing one from scratch, the goal is to adapt an existing solution that aligns with PMWeb's technical requirements and can be integrated efficiently.

Language choice is also a central design consideration. While the PM desktop application uses Object Constraint Language (OCL) to define business rules, the web version will shift to Python. Python was chosen because it has strong tool support and is easier to maintain over time. OCL, on the other hand, doesn't have a solid language server, which made it harder to fit into this system's goals.

With these core decisions in place, the following sections will describe the existing structure of PMWeb and explain how the new LSP component will be integrated into its architecture.

## Architecture

PMWeb is a web-based application designed to manage versioned insurance data, providing insurers with a structured and flexible platform for data modeling and product configuration. The system architecture follows a modular, service-oriented approach, where different components interact to provide a seamless user experience.

At the highest level of abstraction, PMWeb is represented as a single system, interacting with two types of users: registered and non-registered users. Figure 16 - PMWeb System Context (C4 Model - Level 1) presents a context view of the system, highlighting how PMWeb serves as the core platform for insurance data management.

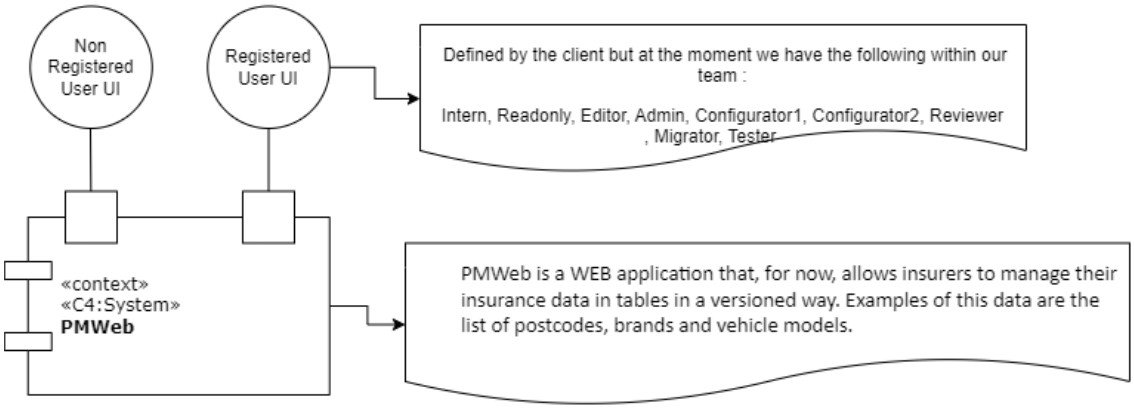



Figure 16 - PMWeb System Context (C4 Model - Level 1)

This diagram illustrates:

- PMWeb as the core system;
- Registered and non-registered users, who access the system through an UI interface;
- User roles, which define access levels within the system;
- The system's primary function until now, which is to manage insurance data in a structured, versioned manner.

At a more detailed level, PMWeb consists of multiple independent services, each deployed as a separate Docker container, ensuring modularity and scalability. Figure 17 - PMWeb Physical Architecture (C4 Model - Level 2) provides a container-level representation of PMWeb's architecture.

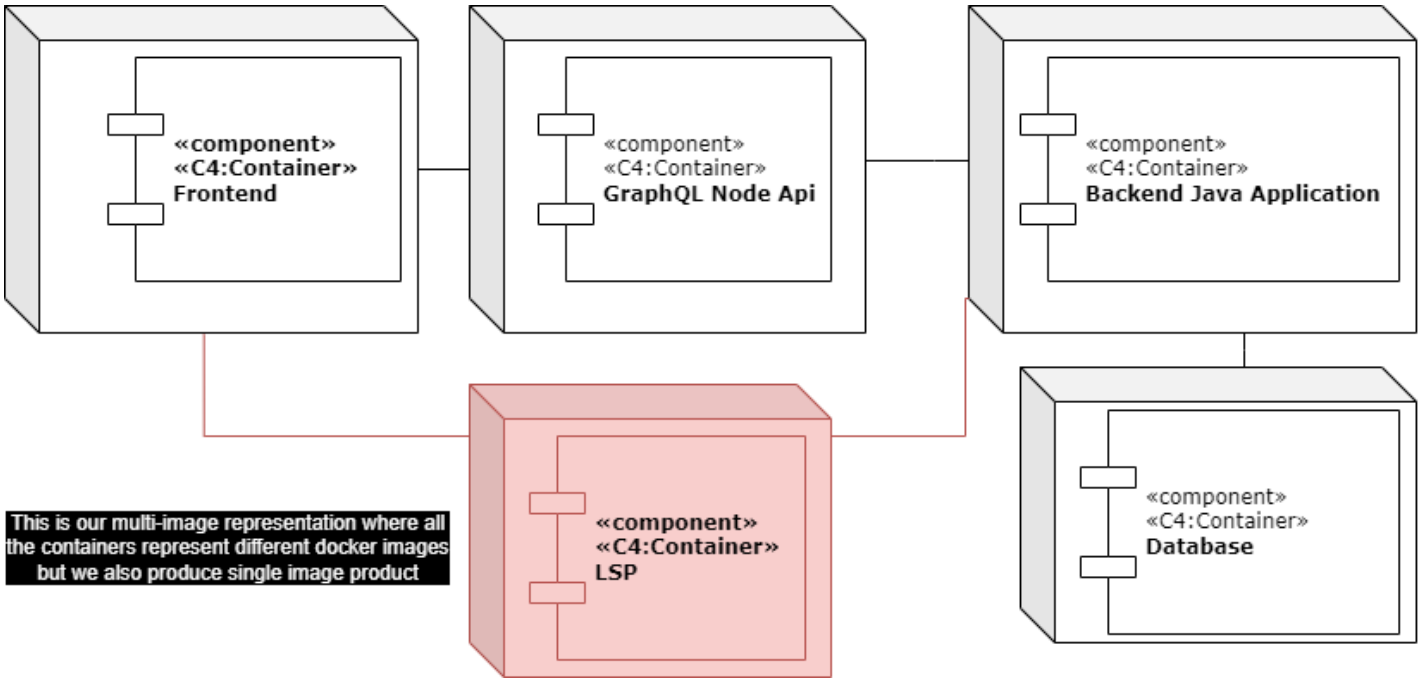



Figure 17 - PMWeb Physical Architecture (C4 Model - Level 2)

This architecture consists of the following main components:

- **Frontend**: A React-based web application, responsible for rendering the user interface and integrating with Monaco Editor;
- **GraphQL Node API**: A GraphQL-based Node.js service (Apollo Client and Apollo Server) that serves as an intermediary between the frontend and backend;
- **Backend Java Application**: A Java Spring application, which executes business logic and interacts with the database;
- **Database**: A Dolt version-controlled database, enabling structured and trackable changes to insurance data.

### The New LSP Component in PMWeb

A new component, the **Language Server Protocol (LSP)**, has been introduced into PMWeb's architecture. This addition is highlighted in red in all architectural diagrams, indicating its role as a new service that must be integrated into the existing system.

The LSP is responsible for handling Monaco Editor requests, providing real-time syntax validation, autocompletion, and formula validation. As Figure 18 - LSP Integration in PMWeb (C4 Model - Level 2) illustrates, it interacts with multiple system components, ensuring seamless integration into the PMWeb workflow.

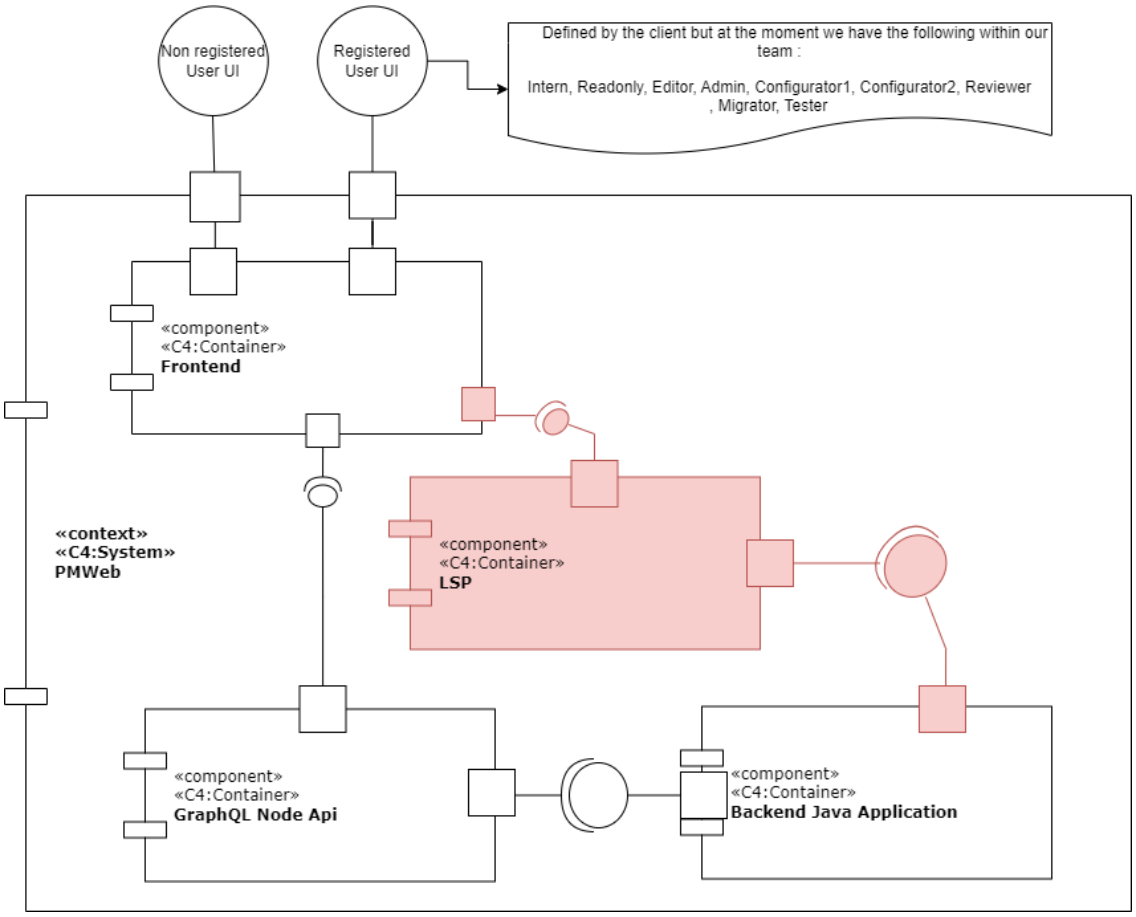



Figure 18 - LSP Integration in PMWeb (C4 Model - Level 2)

The LSP's integration flow consists of the following key interactions:

1. Frontend Integration (Monaco Editor & LSP Communication)

   - The Monaco Editor, embedded in the React Frontend, sends requests to the LSP for real-time validation and autocompletion;
   - Communication between Monaco and the LSP is established via JSON-RPC over WebSockets, ensuring low-latency, bi-directional communication.

2. Backend Integration for Formula Retrieval (Planned)

   • In future iterations, the LSP will be extended to retrieve pre-existing formulas directly from the backend;

   • This will enable support for cross-formula references (e.g., Formula X referencing Formula T), allowing the validation engine to reason across dependencies;

   • By resolving these references at validation time, the system can prevent errors caused by missing or undefined formulas.

Now that the system's structure is defined, the following Figure 19 - Formula Editing Sequence Diagram illustrates the sequence of interactions between components when the user engages with this functionality.

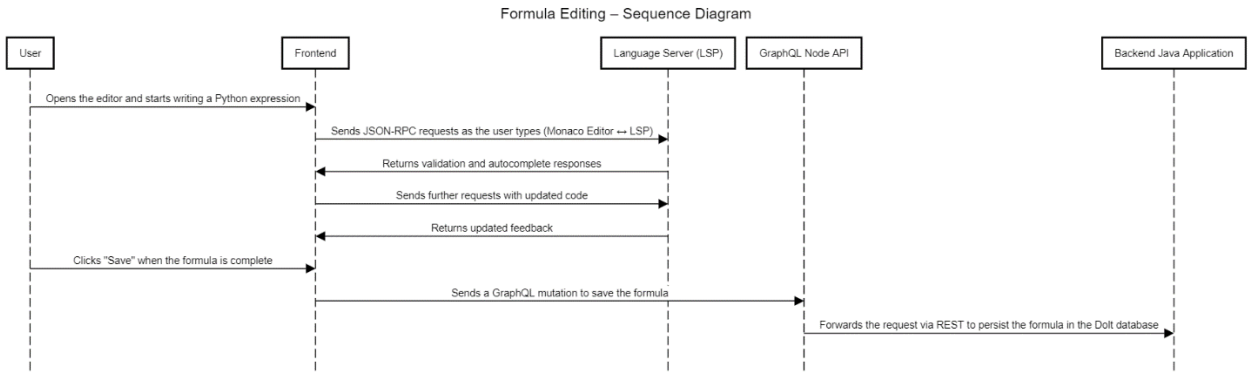



Figure 19 - Formula Editing Sequence Diagram

Using a structured C4 modeling approach, we have clearly defined how the LSP integrates into the existing system without disrupting core functionality.

With this LSP design and future implementation, PMWeb is positioned to provide a modern, efficient, and intuitive formula editing experience, allowing insurers to streamline their workflows with greater accuracy and automation.

## 5.2 Requirement 2: Business Object Model (BOM) Integration

To support domain-specific validation and contextual autocompletion during formula editing, it is essential that the Language Server Protocol (LSP) component used in PMWeb understands the structure and semantics of the domain entities involved in insurance product modeling. This is achieved by integrating the Business Object Model (BOM) directly into the LSP's workspace context.

### Motivation and Design Objective

The BOM describes key business entities such as Person, Address, or Contract, and their attributes, types, and relationships. In the PMWeb desktop predecessor, these objects were natively integrated into the modeling environment. To replicate this experience in the web-based PMWeb, the BOM must become a **first-class citizen in the editor**, enabling formulas to reference and validate against business entities, for example, allowing a formula context like self: Address and supporting attribute access like Address.city.

The Business Object Model (BOM) in PMWeb is defined using **JSON Schema**, a format that describes structured data through object types, fields, and constraints. This choice was made

at the company level. It offers a standard way to represent business models in a platform-agnostic format, aligning with the organization's broader technology strategy.

However, the Monaco Editor and the LSP (based on Pyright) cannot interpret a JSON structure directly. Pyright, as a static type checker for Python, requires Python code to perform syntax and semantic analysis. Therefore, a key architectural requirement was to bridge the gap between the *JSON-based representation of the BOM* and the *Python-based analysis engine* in the LSP.

## Architecture and Workflow

To integrate the BOM in this JSON Schema format with the Python-based Language Server Protocol (LSP), the workflow proceeds as follows:

1. **Receiving the BOM**:
   Each client provides a BOM definition in the form of a JSON Schema file. This file defines the domain objects and their structure for a given product or data version. The file is made available to the LSP.

2. **Transforming JSON to Python**:
   Upon initialization of the LSP container, a dedicated transformation script/function is executed. This script/function reads the BOM JSON and transforms it into a format that the LSP can validate formulas against like it is described in the Figure 20 – BOM Integration.

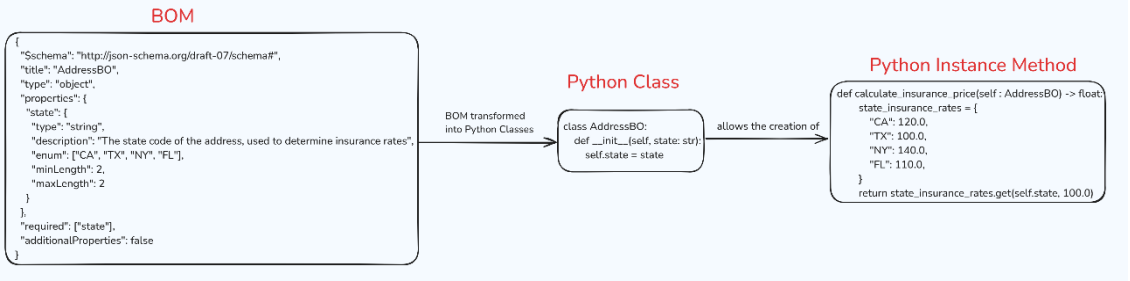



Figure 20 – BOM Integration

3. **Binding the BOM to the Editing Context**:
   Once the BOM is made a part of the LSP, the system then needs to ensure that the editor has access to it in terms of the data version being used. This way, the editor will be able to respond properly to user input, using the BOM to check the code and return proper guidance as it edits.

4. **Enabling Domain-Aware Editing**:
   The editing experience should be enriched with contextual knowledge derived from the BOM, allowing the system to provide relevant suggestions, validations, and feedback based on the user's current domain version.

A summary of this workflow is illustrated in Figure 21 - High-Level Flow for BOM Integration into the LSP, which shows the full flow from BOM loading and transformation to real-time validation within the formula editor.

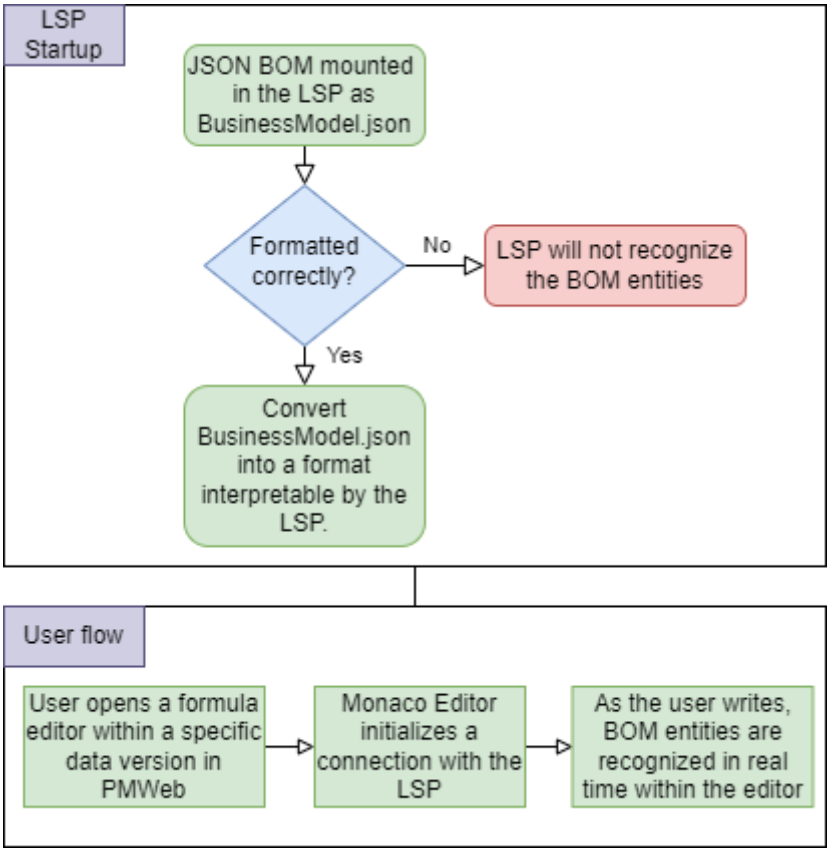



Figure 21 - High-Level Flow for BOM Integration into the LSP

### Design Considerations

This architecture was chosen based on the following considerations:

- **Separation of Concerns**: The BOM logic is decoupled from the LSP itself. Pyright remains a standard LSP engine, and all domain knowledge is injected externally via JSON Schema files;
- **Reusability and Maintainability**: The transformation logic from JSON to Python can be reused across clients and adapted to other domains if needed.

## 5.3 Metadata Design for Formula Persistence

To support formula editing as a first-class feature within PMWeb, it was necessary to introduce a new metadata structure that formally represents what a formula is, how it behaves, and how it relates to the business model. This design establishes the foundations for storing, validating, and executing formulas across different product configurations, while supporting collaborative modeling via Dolt's branching mechanism.

A formula in this system is treated as a reusable logic unit that is always tied to a specific context, typically a Business Object Model (BOM) entity. For instance, if a formula is defined in the context of a Person object, it behaves as if it were written inside that class. In Python, this would correspond to using a self: Person annotation within the expression.

Each formula includes a unique name, a description, and a return type. The return type can be any valid Python type or a custom type defined in the BOM. Formulas are also assigned a language. While Python is the only supported language for now, the system is designed to support additional languages in the future.

Given this need for structure, a new set of relational tables was created to persist all formula-related metadata. These tables are stored and versioned using Dolt, meaning each branch of a product or model can have its own set of formulas, variations, and parameters, enabling *real-time collaborative modeling* across business units.

### DWB_FORMULA

This table captures the base metadata for each formula:

```
CREATE TABLE DWB_FORMULA (
  ID varchar(8) NOT NULL,
  NAME varchar(255) NOT NULL,
  CONTEXT varchar(255) NULL,
  DESCRIPTION varchar(255) NOT NULL,
  RETURN_TYPE varchar(255) NOT NULL,
  FORMULA_LANGUAGE varchar(255) NOT NULL,
  PRIMARY KEY (ID)
);
```

Each entry here defines a formula's identity, purpose, and connection to the model. The CONTEXT attribute relates the formula to the domain object it operates on, which can be used for autocompletion, scoping, and validation.

### DWB_FORMULA_VARIATION

Currently, each formula in PMWeb has a single expression, but to support future requirements such as variation per product or conditional logic switching, formulas are designed to support

multiple *variations*. Each variation contains a named expression that implements the formula's logic:

```
CREATE TABLE DWB_FORMULA_VARIATION (
  ID varchar(8) NOT NULL,
  NAME varchar(255) NOT NULL,
  EXPRESSION varchar(4000) NULL,
  FORMULA_ID VARCHAR(8),
  PRIMARY KEY (ID)
);
FOREIGN KEY (FORMULA_ID) REFERENCES DWB_FORMULA (ID)
```

This separation ensures that business users can define and experiment with alternative implementations of the same logic unit, enabling more dynamic modeling.

### DWB_FORMULA_PARAMETERS

Formulas may require input parameters. To support this, parameters are modeled explicitly, with name, type, and order:

```
CREATE TABLE DWB_FORMULA_PARAMETERS (
  ID varchar(8) NOT NULL,
  NAME varchar(255) NOT NULL,
  PARAM_TYPE varchar(255) NOT NULL,
  PARAM_ORDER INT,
  FORMULA_ID VARCHAR(8),
  PRIMARY KEY (ID)
);
FOREIGN KEY (FORMULA_ID) REFERENCES DWB_FORMULA (ID)
```

The PARAM_TYPE supports both native Python types and BOM-defined types, which will allow future extensions such as complex object input or nested validations.

### DWB_FORMULA_INVOCATION

To enable traceability and analysis of formula dependencies, such as which formulas call others, a dedicated invocation table was introduced:

```
CREATE TABLE DWB_FORMULA_INVOCATION (
  ID varchar(8) NOT NULL,
  CALLER_FORMULA_VARIATION_ID VARCHAR(8),
  CALLED_FORMULA_ID VARCHAR(8),
  PRIMARY KEY (ID)
);
FOREIGN KEY (CALLER_FORMULA_VARIATION_ID) REFERENCES
DWB_FORMULA_VARIATION (ID)
FOREIGN KEY (CALLED_FORMULA_ID) REFERENCES DWB_FORMULA (ID)
```

This mechanism enables future static analysis tools to validate dependency graphs and detect issues like circular references or unintended side effects.

### Branching and Collaboration

Because PMWeb relies on *Dolt* as its database engine, all metadata tables described above are version controlled. Each branch in Dolt represents a separate modeling context, for example, a product version or a work-in-progress feature. With this setup, teams can work on different versions of formulas without stepping on each other's changes. Updates can be reviewed, merged, and traced back to the person who made them.

Formulas are no longer just embedded pieces of code, they're handled as standalone, versioned components that teams can update and reuse over time. This also opens the door to stronger tool support and deeper integration with the Language Server Protocol (LSP).

## 5.4 CI/CD Pipelines and Delivery Strategy for the LSP Component

To support the development and deployment of the new LSP microservice, CI/CD processes have been set up using Jenkins, with integrations to GitLab, SonarQube, and Dependency-Track. This setup allows the PMWeb engineering team to automate testing, perform security checks, and generate deployment artifacts, making it easier to deliver the LSP reliably across both internal environments and client-facing systems.

The CI/CD process is split into two pipelines. One focuses on validating code quality and running automated checks, while the other handles the creation and distribution of Docker images.

### Multi-Branch Validation Pipeline

The first pipeline is configured as a *multi-branch Jenkins pipeline*, triggered automatically whenever a new branch is pushed to the GitLab repository. This setup allows for parallel validation across multiple branches, supporting a fast-paced development workflow.

This pipeline ensures that all feature branches undergo the same quality gates, making it easier to detect and resolve issues early in the development cycle.

### Docker Image Build and Delivery Pipeline

The second pipeline is triggered when a branch is merged into the main development branch (*develop*). It starts only if the first pipeline passes all checks, ensuring that only validated and secure code proceeds with packaging.

This separation between validation and packaging ensures traceability and control over what is deployed or released.

### Importance of the Pipeline Strategy

This pipeline strategy ensures:

- *Continuous integration* with consistent validation of new code;
- *Automated testing and static analysis* to prevent regressions;
- *Proactive vulnerability detection* in both application code and dependencies;
- *Reliable Docker image creation* ready for delivery and deployment.

These pipelines are a vital part of the system's overall design, reinforcing the maintainability, security, and quality of the LSP component through automation.

# 6 Implementation

This chapter presents the implementation of the formula editing and validation system defined during the design phase. It includes the development of a custom Language Server Protocol (LSP) server for Python, the integration of this server with the Monaco Editor in the frontend, and the automation pipelines that support testing and delivery.

At its core, the solution enables real-time syntax validation, autocompletion, and static analysis for user-defined formulas in PMWeb. These capabilities are provided through a web-based editor that communicates with a backend LSP service using the JSON-RPC protocol over WebSockets.

To ensure compatibility with PMWeb's architecture and development practices, the backend was implemented in *TypeScript* using *Node.js*, *Express.js*, and the open-source *Pyright* static type checker. On the frontend, a reusable React component embeds Monaco Editor and connects to the LSP using the `monaco-languageclient` library. The implementation draws inspiration from the monaco-languageclient project by TypeFox [33] but was customized extensively to handle domain-specific requirements and user-defined metadata.

In addition to the core functionality, a CI/CD pipeline was implemented using *Jenkins* to automate validation and deployment. This includes two pipelines: one for validating changes across branches (with unit testing, static code analysis, and dependency checks), and another for building and distributing Docker images to internal and client environments. Together, these pipelines ensure that the language server can be deployed consistently, securely, and with high confidence.

The following sections provide a detailed overview of the backend and frontend components, the integration of domain-specific metadata into the LSP environment, and the build and deployment strategies that support continuous delivery.

The implementation consists of two core components:

- A *custom LSP server*, built in Node.js, that wraps Pyright and handles LSP requests over WebSocket;
- A *React frontend component*, named PythonCodeEditor, which embeds Monaco Editor and communicates with the backend using the LSP protocol.

These two components form the core of the formula editing experience. Users interact with a code editor that feels like a lightweight IDE, with real-time feedback powered by the LSP server.

## 6.1 Backend: Node.js LSP Server with Pyright

### Objectives and Setup

The primary goal of the backend was to implement a lightweight, stateless Language Server Protocol (LSP) server that integrates seamlessly with PMWeb's web-based frontend. Rather than building a language server from scratch, the solution leverages *Pyright*, an open-source static type checker for Python developed by Microsoft. Pyright is written in TypeScript and can expose its language features via a JSON-RPC interface, making it compatible with Monaco Editor through the LSP standard.

The LSP is implemented in *TypeScript* and structured as a standalone Node.js service. It includes the following core components:

- **Express HTTP Server**: Provides the application entry point and basic HTTP capabilities (e.g., health checks or stub endpoints);
- **WebSocket Server**: Uses the ws library to handle JSON-RPC connections and forward them between the client and the Pyright process;
- **Pyright Process Management**: The Pyright language server is launched as a subprocess, and its standard input/output streams are connected to the WebSocket using the vscode-ws-jsonrpc bridge;
- **Environment Configuration**: The service is fully configurable via .env files and utilities like getRequiredEnv to inject runtime parameters such as the server port and backend URLs;
- **Modular Controller Architecture**: To accommodate future extensions such as formula dependency analysis or BOM-aware completions, the backend is organized into modular controller and service layers.

### Core Implementation

The main entry point for the backend is src/app.ts. This file is responsible for starting the server, resolving runtime paths, initializing logging, and handling WebSocket upgrades.

- Loads environment variables (e.g., PMW_LSP_PYTHON_PORT);
- Sets up an Express HTTP server and upgrades incoming connections to WebSocket;
- Loads the Pyright language server from the path node_modules/pyright/dist/pyright-langserver.js;
- Spawns the Pyright process and connects it to the WebSocket stream using vscode-ws-jsonrpc and vscode-languageserver.

A simplified excerpt of the WebSocket-to-LSP bridge is shown below:

```
const socket = createWebSocketConnection(req, socket, head)
const reader = new WebSocketMessageReader(socket)
const writer = new WebSocketMessageWriter(socket)

const pyrightPath = resolve(__dirname,
'../node_modules/pyright/dist/pyright-langserver.js')
const pyrightProcess = spawn('node', [pyrightPath, '--stdio'])

forward(reader, pyrightProcess.stdin)
forward(pyrightProcess.stdout, writer)
```

This pattern effectively transforms the server into a transparent proxy that routes LSP messages between Monaco Editor and Pyright in real time. This architecture is directly inspired by the TypeFox monaco-languageclient example for Python [34] but adapted to support PMWeb's modular infrastructure and runtime constraints.

### Injecting BOM into the LSP

While the core of the implementation focuses on enabling Python-based formula validation, the long-term goal is to create an intelligent, domain-aware editing environment. This requires integrating PMWeb's internal Business Object Model (BOM) into the language server environment so that formulas referencing business entities are recognized and validated just like native Python types.

To achieve this, a dedicated process was implemented to transform PMWeb's business modeling metadata into a Python-compatible format that the LSP (Pyright) can understand and analyze. The BOM is currently defined as a static JSON Schema file (businessModel.json) located in the project's model/ folder. When the LSP is deployed as a Docker container, this file can be mounted by clients into the container's filesystem, making the BOM accessible for interpretation and validation.

To convert the JSON definition into something usable by Pyright, a transformation step is executed during LSP container startup. The function createModelStubFiles() is responsible for this transformation. It parses the JSON BOM and generates a Python module (BusinessModel.py) that contains typed class definitions matching the structure described in the JSON. These

definitions include properties, methods, and type annotations compatible with Pyright's static analysis engine as displayed in Figure 22 - Result of createModelStubFiles() on a Model Object called PropertyBO.

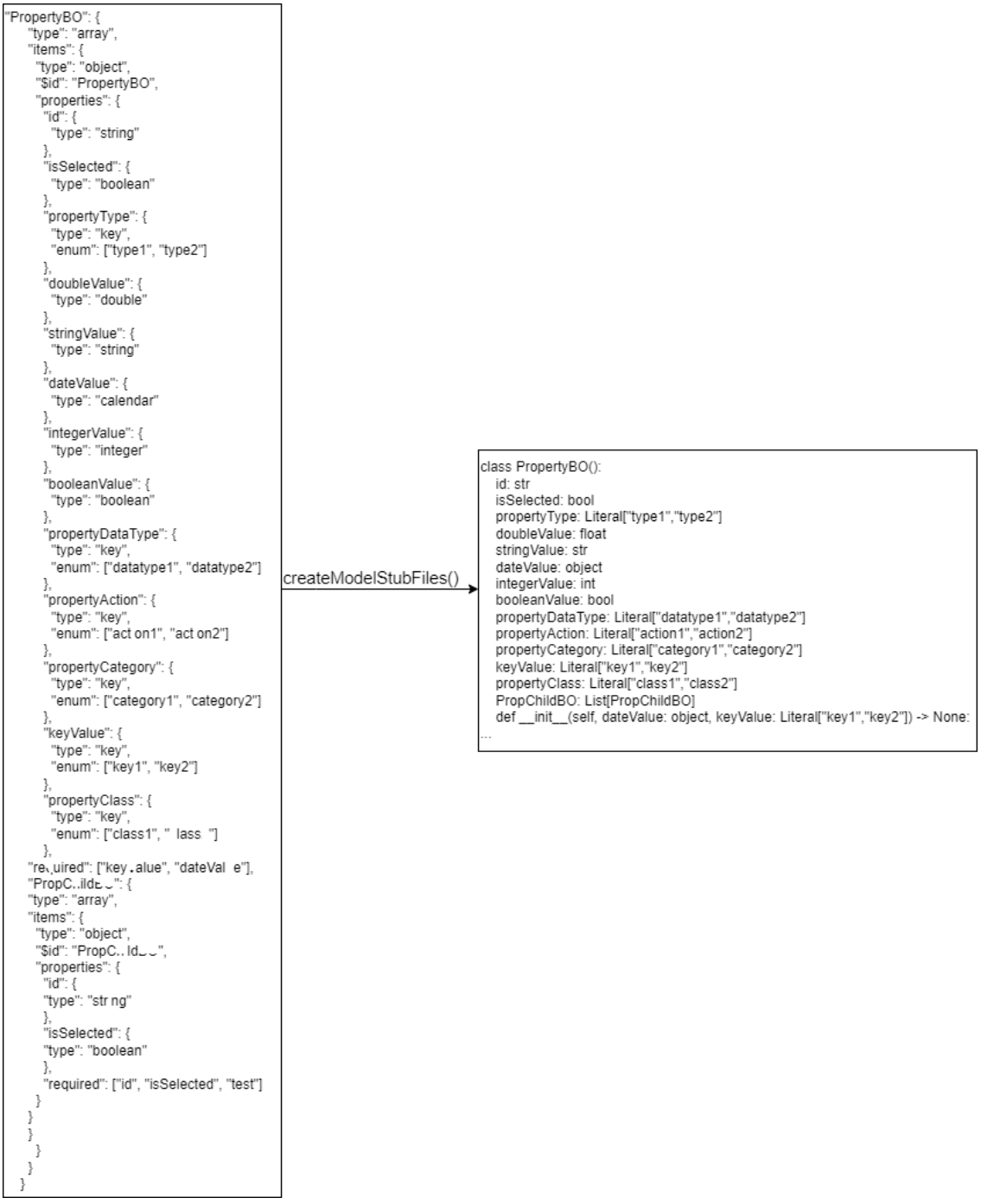



Figure 22 - Result of createModelStubFiles() on a Model Object called PropertyBO

### Why Workspace Configuration Matters

For Pyright to resolve these generated modules and provide accurate validation, its workspace context must include the folder where the BOM file resides. This is why the frontend explicitly sets a workspaceFolder when initializing the MonacoLanguageClient, pointing to the version-specific directory where BusinessModel.py lives.

This behavior aligns with Pyright's design expectations. Unlike dynamic interpreters, Pyright relies on a static file-based workspace and does not support implicit runtime environments. By mapping the editor session to a simulated filesystem that includes both the formula, and its domain dependencies, full validation and type inference become possible.

Each time a user opens the formula editor within a specific data version, the frontend establishes a new WebSocket connection to the LSP, which is scoped to a dedicated folder created inside the LSP container. This folder serves as the workspace for that session and contains the generated BusinessModel.py file derived from the BOM. Although the formula being edited by the user is not stored as a physical file, it is injected into the LSP's in-memory workspace representation, allowing Pyright to treat it as part of the same environment. If a workspace folder for that version already exists, it is reused to avoid unnecessary regeneration.

To ensure that the business objects are available within the formula context, the editor should contain the line **BusinessModel import *** to the user's code. Since the LSP is operating within the same workspace directory, it can resolve this import seamlessly. This setup enables syntax checking, autocomplete, and type-aware tooltips in Monaco Editor based on the domain structure defined in the BOM.

This entire structure is version sensitive. Each workspace is tied to a specific data version, meaning that in the future, any change to the BOM, such as adding a new field or renaming an object, affects only the associated context. This ensures isolation between data versions, preventing regressions in unrelated formulas and mirroring the behavior of the version-controlled Dolt database. It also makes it possible to evolve domain models incrementally without breaking existing user workflows.

This process lays the foundation for future enhancements such as:

- Allowing users to define BOM objects dynamically through a web interface;
- Supporting validation across versions by diffing or merging BOM definitions across data branches.

Together, these steps transform the LSP into a fully domain-aware assistant, capable of reasoning about both Python semantics and PMWeb-specific modeling constructs.

## 6.2 Frontend: PythonCodeEditor Component

To provide a seamless and intelligent editing experience for formulas within PMWeb, a dedicated React component named *PythonCodeEditor* was implemented. This component embeds the Monaco Editor and establishes a live WebSocket connection to the custom LSP server built with Pyright, described in the Section Backend: Node.js LSP Server with Pyright. The frontend implementation was strongly inspired by the TypeFox monaco-languageclient example for Python [34] but extended significantly to meet the requirements of a multi-user, domain-aware modeling environment.

### Key Responsibilities

The PythonCodeEditor component is responsible for:

- Initializing the Monaco Editor using a customized setup based on @codingame/monaco-vscode-* overrides;
- Registering an in-memory file system to allow Pyright to perform validation on virtual documents;
- Injecting necessary domain context (imports, function headers) into the code model;
- Managing WebSocket-based communication with the backend LSP server;
- Displaying diagnostics and suggestions in real time, with visual markers inside the editor;
- Preventing edits to fixed structural blocks (like imports or headers) and enforcing formula size limits.

### Enhancing Monaco for Domain-Specific Modeling

Since PMWeb's formulas rely on business-specific models, the editor environment must simulate a real Python workspace that knows about these types. This is achieved by:

- Automatically injecting base imports like from BusinessModel import *. This import is responsible for enabling the recognition of BusinessModel objects as part of the language, so the syntax validation stays clean;
- Dynamically constructing the function signature using metadata from the formula object (e.g., name, parameters, return type);
- Adding Monaco decorations to lock specific lines and prevent users from modifying the injected structure;
- Custom logic is added to prevent pasting content that exceeds the formula character limit (MAX_CHARS), improving error handling and guidance.

We can see those enhancements in the Figure 23 - Layout of the editor with locked header lines and formula definition.

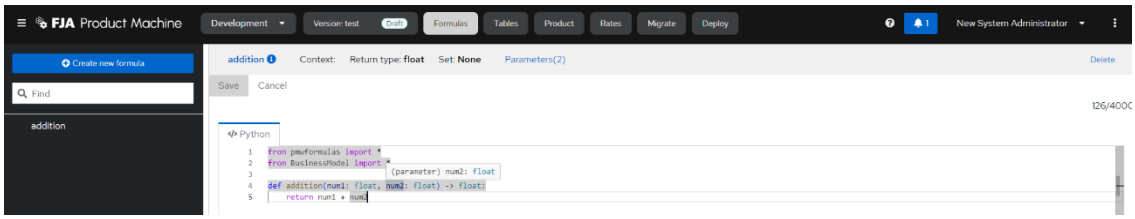


Figure 23 - Layout of the editor with locked header lines and formula definition

This structure ensures that the Pyright LSP receives syntactically and semantically valid code for analysis, even though the code is written in-browser and is never saved to disk.

### Communication with the LSP

Communication with the LSP server is achieved using WebSocket and the JSON-RPC protocol. On editor initialization:

1. The editor registers an in-memory file (code.py) using a virtual file system (fileSystemProvider);
2. The client opens a WebSocket connection to the Node.js backend (lsp-python/ws);
3. Upon connection, a MonacoLanguageClient is initialized, and the document is opened and synchronized with Pyright.

This communication loop ensures that any change in the editor is immediately validated, and that diagnostics are shown in real time using Monaco's marker APIs.

The LSP client is initialized with workspace-level configuration, including the virtual workspace URI and editor settings like theme and font size. This setup mimics VS Code behavior as closely as possible.

### Code Snippets and Comparison

A small excerpt from the initLanguageClient function illustrates the initialization of the MonacoLanguageClient with error handling and workspace context:

```
languageClient = new MonacoLanguageClient({
  name: 'Pyright Language Client',
  clientOptions: {
    documentSelector: ['python'],
    workspaceFolder: {
      index: 0,
      name: 'workspace',
      uri: monaco.Uri.parse(workspacePath)
    },
    errorHandler: {
      error: () => ({ action: ErrorAction.Continue }),
      closed: () => ({ action: CloseAction.DoNotRestart })
    }
  },
```

```
  connectionProvider: {
    get: () => Promise.resolve({ reader, writer })
  }
})
```

This closely mirrors the setup from the TypeFox Pyright example [34], but customized with in-memory file handling, formula injection, and workspace overrides tailored to PMWeb's domain.

### Saving Formulas and Leveraging Dolt for Collaboration

Once the user finishes editing, creating, or removing a formula, the UI provides a straightforward action to save the change. This triggers a GraphQL mutation that sends the formula's content, held entirely in memory during the editing session, to the Node.js GraphQL API, which then communicates with the Java backend. There, the change is persisted within the currently active Dolt data version.

Each formula is always tied to a specific data version, and thanks to Dolt's version-controlled architecture, no additional complexity is required to support collaborative editing or branching. When a formula is saved, its changes are scoped to that data version: new formulas are tracked as **additions**, modified ones as **diffs**, and removed formulas are automatically tracked as **deletions**. Once the data version is approved through the platform's version management flow, all changes, additions, updates, and removals, are merged into the main branch. From that point on, the official model reflects the updated formula landscape: new entries appear, edited ones show updated content, and deleted ones are no longer present.

This behavior emerges naturally from Dolt's design, which treats the database like a Git repository. Because Dolt was selected early in the project, this entire workflow, from isolated editing to branching, merging, and versioning, was already supported at the persistence layer. No additional infrastructure was needed to achieve granular tracking, history, or branching of formulas. Dolt fit the needs of the application seamlessly, effectively enabling a collaborative, version-aware modeling environment out of the box.

## 6.3 CI/CD Pipeline Implementation

To ensure code quality, security, and consistent delivery of the new LSP component, a two-stage CI/CD pipeline was implemented using Jenkins as it was discussed on the section CI/CD Pipelines and Delivery Strategy for the LSP Component. These pipelines automate validation, testing, image building, and deployment. They are shared across the PMWeb Node.js monorepo, which includes the frontend, GraphQL API, and the new Python LSP service.

### Multi-Branch Validation Pipeline

Every time a new branch is pushed, or a merge request is opened in GitLab, a validation pipeline is triggered automatically. This pipeline focuses on code correctness, testing, and dependency security across all shared modules, including the LSP.

The stages of this pipeline represented in Figure 24- Multi-Branch Validation Pipeline for LSP Component are as follows:

1. *Git Checkout*
   Jenkins checks out the specific feature or bugfix branch that triggered the pipeline.
2. *Dependency Installation*
   As the LSP is a Node.js-based service, dependencies are installed using yarn install.
3. *Build Step*
   The application is compiled using yarn build, generating production-ready output.
4. *Unit Tests*
   Automated tests are executed to validate the core logic of the application.
5. *Static Code Analysis*
   SonarQube performs a *static code analysis*, identifying bugs, code smells, and maintainability issues.
6. *Software Composition Analysis (SCA)*
   Dependency-Track is used to perform an SCA scan, identifying known vulnerabilities in third-party packages.
7. Pipeline Status Reporting
   Jenkins integrates with GitLab to report the status of the pipeline directly on the corresponding branch, giving developers immediate feedback on code quality.

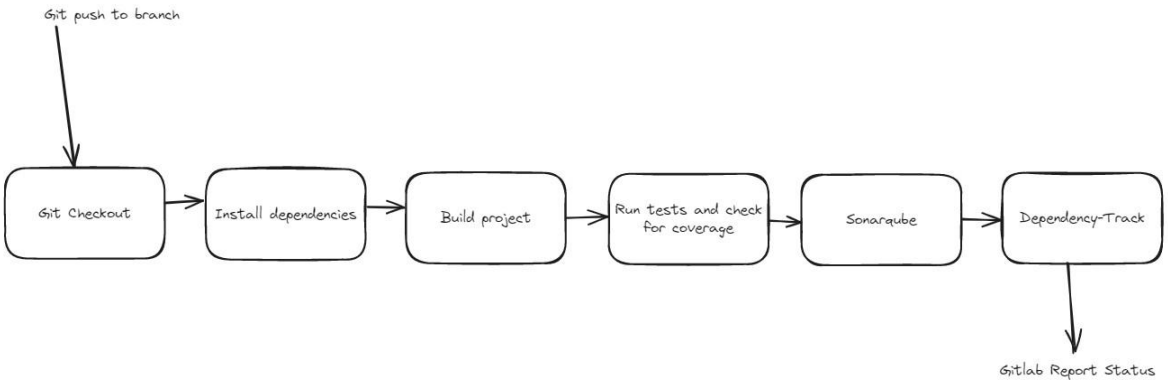



Figure 24- Multi-Branch Validation Pipeline for LSP Component

- **Build and Test**
  The pipeline begins by installing dependencies and building the codebase using Yarn:

```
yarn install
yarn build
yarn stage:coverage
```

This process compiles all shared components and executes unit tests with coverage validation. The version is extracted from lerna.json and used throughout the pipeline.

- **Static Code Analysis with SonarQube**
  A parallel job uses the SonarQube scanner (sonar-scanner-5) to perform static analysis:

```
sh "${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=${projectKey} ..."
```

If the project does not already exist in SonarQube, it is created dynamically and assigned to the "DWB" quality gate.

- **Dependency Track Analysis**
  To ensure third-party libraries are secure, the pipeline generates a Software Bill of Materials (SBOM) using cdxgen:

```
cdxgen -t node.js -o bom.json
```

This SBOM is uploaded to a Dependency Track server. If it's a new version, the pipeline also uploads a VEX (Vulnerability Exploitability Exchange) file to help triage vulnerabilities. The process waits until all vulnerability data is fully processed.

Vulnerability thresholds are strictly enforced:

```
failedTotalCritical: 0,
failedTotalHigh: 0,
failedTotalMedium: 0,
failedTotalLow: 0
```

- **Triggering Docker Builds**
  If the branch is develop or legacy, this pipeline triggers downstream jobs that build Docker images for the frontend, GraphQL API, and LSP:

```
build job: 'pmw.docker.node-lsp-python', parameters: [...]
```


Figure 25- Jenkins validation pipeline run (success state)

### Docker Image Build and Deployment

When changes to the LSP are merged into the develop branch, a second Jenkins pipeline is triggered. This pipeline focuses on packaging the validated code into a Docker image and preparing it for deployment.

The process in Figure 26 - Docker Image Build and Delivery Pipeline is broken down into the following steps:

1. *Git Checkout (develop branch)*
   The pipeline begins by checking out the latest version of the develop branch.

2. *Dependency Installation and Build*
   Dependencies are installed using yarn install, and the build is generated using yarn build.

3. *Docker Build*
   A Dockerfile is used to package the application into a Docker image.

4. *Container Vulnerability Scan*
   A scan is performed on the resulting Docker image to ensure that both the base image and application layers are free from critical vulnerabilities.

5. *Docker Push*
   If the image passes all quality and security checks, it is pushed to the internal Nexus repository, making it available for deployment across environments and for distribution to clients.

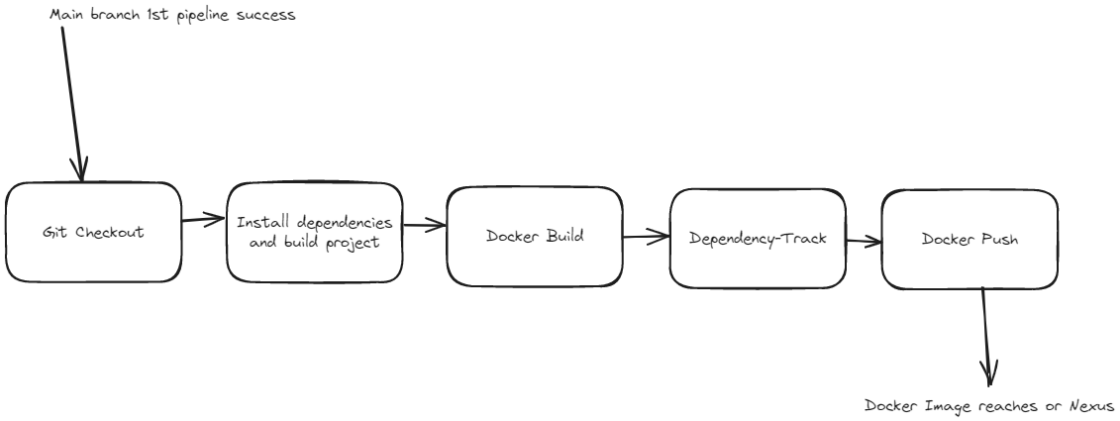



Figure 26 - Docker Image Build and Delivery Pipeline

- **Checkout and Yarn Build**
  The pipeline checks out the latest version of the monorepo and runs a scoped build for the LSP service:

```
yarn build:lsp-python &
yarn install --production
```

After the build, relevant files (code, configs, dependencies) are moved into a folder named files_to_go_into_docker.

- **Docker Image Creation**
  The Dockerfile used is based on a minimal UBI9 Node.js 18 image. It prepares the runtime, installs necessary tools, and sets up the LSP application directory:

```
FROM vg-s-nexus.dslocal.com:5000/ubi9/nodejs-18-minimal
COPY --chown=pmweb-user:root $COPY_SOURCE $PMWEB_HOME/js-monorepo
CMD ["/usr/local/pmweb/js-monorepo/packages/lsp-python/startup.sh"]
```

The startup.sh script handles environment setup and starts the LSP using npm start. It ensures flexibility by checking for environment variables like PMW_LSP_PYTHON_PORT and writing log configuration files if needed.

- **Security and Dependency Analysis**
  As with the validation pipeline, the image undergoes SBOM generation and a second Dependency Track scan. VEX files are uploaded if needed.

- **Push to Nexus Registry**
  Finally, the Docker image is pushed to the internal Nexus Docker registry:

```
dockerImage.push(getImageVersion())
```

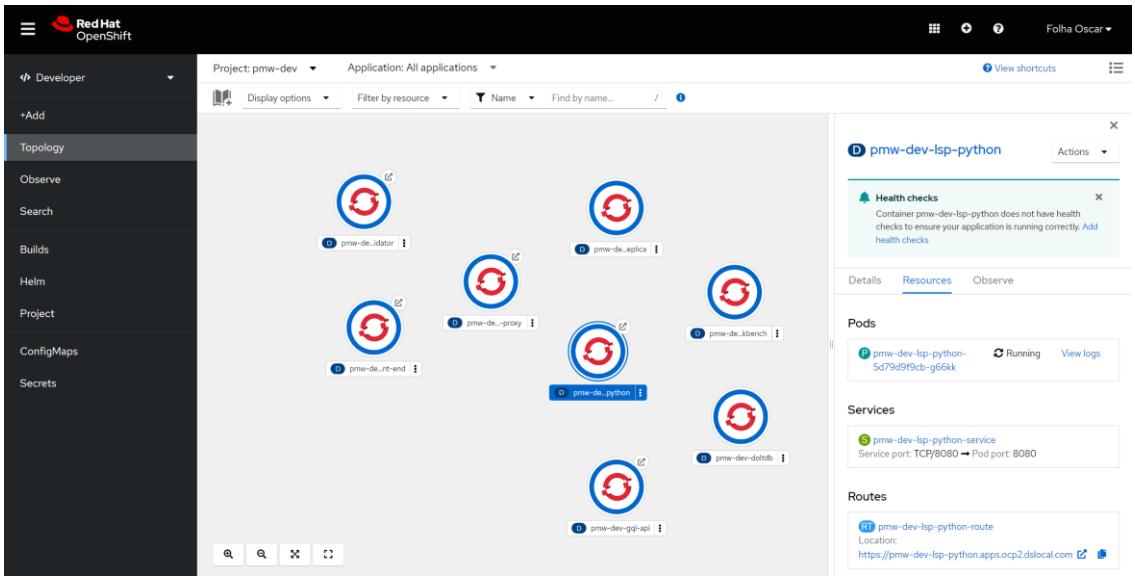



Figure 27- OpenShift pod running the latest LSP image

These two pipelines ensure the LSP component is continuously validated, securely packaged, and reliably deployed. Their integration with GitLab, Jenkins, SonarQube and Dependency Track, reflects a mature DevOps approach and helps maintain high standards across all PMWeb environments.

# 7 Validation

The Validation chapter serves as a critical part of this dissertation, where the effectiveness and robustness of the implemented system are evaluated against various predefined criteria. The validation process not only ensures the functional correctness of the solution but also assesses its ability to perform in real-world scenarios. This chapter provides an in-depth examination of the validation activities carried out during the development process, focusing on both automated and manual testing strategies.

A specific BOM (Business Object Model) file was created for testing purposes to evaluate how the LSP handles complex domain-specific data structures. This BOM file was designed to reflect the AddressBO and its related components, which are crucial for simulating real-world business objects within the project.

The BOM file used for testing purposes has the following structure:

- **Root Object: AddressBO**
    - **Properties:**
        - id: A string representing the unique identifier for the address.
        - AddressLine1, AddressLine2, City, ExternAddrId, PostCode, Suburb, Town: Strings representing various address fields.
        - AddressType, AddressUsage, State, Country, AddressCategory, AddressClass: Enums representing different address classifications, with predefined values like Mailing, Billing, VA, IL, etc.
        - PropertyBO: An array of PropertyBO objects, which represent individual properties associated with the address.
- **Nested Object: PropertyBO**
    - **Properties:**
        - id: A string representing the unique identifier for the property.
        - isSelected: A boolean indicating whether the property is selected.
        - propertyType: Enum representing the type of property, such as type1, type2.
        - doubleValue, stringValue, dateValue, integerValue, booleanValue: Various property values with their respective data types.
        - PropChildBO: An array of PropChildBO objects (nested within PropertyBO), each containing:
            - id: A string.
            - isSelected: A boolean.

        - A required test field (as an example of the structure for child objects).
- **Nested Object: PropertyGroupBO**
    - **Properties:**
        - id: A string representing the unique identifier for the property group.
        - propertyGroupType, propertyGroupClass, propertyGroupCategory: Enums that classify the property group.

And can be visualized in the Figure 28 – Visual Representation of the BOM used in testing.

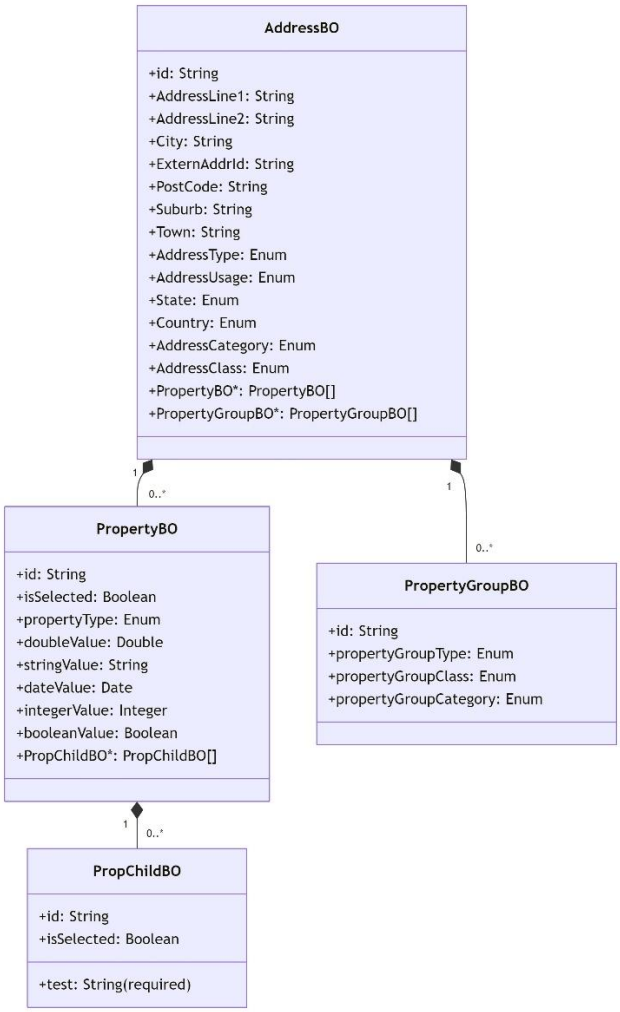



Figure 28 – Visual Representation of the BOM used in testing

Each of these objects was tested within the context of the LSP to ensure proper type recognition, code completion, diagnostics, and signature help. The validation process focused on how well the LSP could infer types, identify issues in the business object structure, and provide useful coding support (e.g., hover information, completion suggestions) in scenarios involving complex types like AddressBO and PropertyBO.

By using this BOM file for testing, we were able to simulate realistic coding scenarios within the domain and verify the LSP's performance and accuracy when dealing with nested, complex data structures. The results of these tests are discussed in the following sections.

## 7.1 Automated Testing with the LSP

To assess the performance and reliability of the Language Server Protocol (LSP) integration, a set of automated tests was conducted. A **Node.js-based script** was developed specifically to interact with the LSP server powered by Pyright. The script was designed to test the LSP server's responses to a series of common coding scenarios. These tests focused on verifying the server's capability to provide accurate type inference, code completions, diagnostics, and other developer tools within a single-file workspace.

The LSP server was hosted locally and accessed via a WebSocket connection at ws://localhost:30001/lsp-python/ws. The workspace was fixed to a well-defined directory within the project's source tree.

This workspace contained all the relevant Python files, including domain-specific business objects (BOs) such as AddressBO and PropertyBO, along with test scripts embodying various code patterns.

### Automated Testing Using a Node.js Script

This script handled:

- Establishing the WebSocket connection to the LSP server;
- Sending workspace initialization and document open notifications;
- Issuing requests for hover, code completion, signature help, diagnostics, and document symbols;
- Measuring response times to evaluate performance;
- Logging and summarizing results for analysis.

By automating these actions, the script enabled repeatable, systematic testing of the LSP server's core features in realistic coding scenarios.

### Test Scenarios

The test cases were carefully designed to cover essential LSP features and common coding situations, including:

- **Type Recognition on Domain Objects**
  Hover requests on self-parameters typed as AddressBO or PropertyBO to validate correct type inference.
- **Completion for Built-in Types**
  Code completions on lists and simple functions to verify suggestion accuracy and completeness.
- **Diagnostic Reporting**
  Scenarios with missing return statements, incorrect return types, and syntax errors to check diagnostic precision.
- **Signature Help and Document Symbols**
  Requests to ensure proper function parameter hints and symbol listing.

Each scenario tested the LSP server's ability to handle Python code semantics, error detection, and editor assistance features within a single file. The request types used on this automated testing are explained within the subsection JSON-RPC.

## Results and Discussion

| Scenario | Request Type | Response Time (ms) | Result Summary |
|---|---|---|---|
| AddressBO self parameter hover | **textDocument/hover** | **3** | **Hover contents: (parameter) self: AddressBO** |
| Simple function completion suggestion | **textDocument/completion** | **9** | **16 completion items** |
| Function Missing Return hover | **textDocument/hover** | **2** | **Hover contents: (function) def foo() -> int** |
| List completions suggestions | **textDocument/completion** | **13** | **52 completion items** |
| Function returns wrong type | **textDocument/hover** | **1** | **Hover contents: (function) def broken() -> int** |
| Function returns right type | **textDocument/hover** | **2** | **Hover contents: (function) def correct() -> str** |
| PropertyBO self parameter hover | **textDocument/hover** | **2** | **Hover contents: (parameter) self: PropertyBO** |
| Signature help | **textDocument/signatureHelp** | **4** | **Signature help with 1 signature(s)** |
| Document symbols | **textDocument/documentSymbol** | **2** | **Document has 3 symbol(s)** |

Table 4 - Automated script LSP test results

The LSP server demonstrated robust performance across all tested scenarios based on Table 4 - Automated script LSP test results. Key observations include:

- **Fast and Reliable Initialization**
  The server initialized quickly with minimal latency, allowing immediate processing of source files.
- **Accurate Type Inference and Hover Information**
  The server consistently returned precise hover information, correctly identifying complex domain types such as AddressBO and PropertyBO within 1–3 ms response times.
- **Effective Code Completions**
  Completions for built-in types returned meaningful and extensive suggestions (up to 52 items), with average response times under 15 ms, facilitating rapid developer assistance.
- **Comprehensive Diagnostic Feedback**
  Errors including missing returns, type mismatches, and syntax faults were detected and reported with clarity, improving code correctness and developer awareness.
- **Support for Signature Help and Document Symbols**
  Function signature hints and document symbol listings were returned accurately and swiftly, enhancing interactive coding experiences.
- **Consistent Low Latency**
  Most requests responded in under 15 ms, validating the LSP's suitability for real-time editor integration.

Overall, these results confirm that the Pyright-powered LSP server provides a responsive and precise environment for Python code analysis and editing support in the context of this project.

The systematic testing approach applied in this study, facilitated by a Node.js automation script, effectively measured the LSP server's performance and correctness. The evidence supports the server's capability to enhance developer productivity through accurate type checking, code completions, diagnostics, and interactive assistance with low latency. These characteristics are critical for embedding the LSP in production-grade development tools tailored to complex business domains.

## 7.2 Manual Testing

Alongside the automated tests, quality assurance (QA) testing was conducted to evaluate the system's stability, reliability, and usability under realistic conditions. This process included functional, integration, and system-level testing, with the goal of validating how well the formula editor and its underlying Language Server Protocol (LSP) performed during common

modeling tasks. The QA team tested various interaction flows, verified output correctness, and ensured that no regressions were introduced during feature development.

The following test cases illustrate key functionalities that were validated during this phase. The images shown were provided by the QA team.

### Autocompletion Based on Formula BOM Context

One of the most important tests involved validating whether the LSP could recognize the formula's BOM context and correctly suggest its attributes. When a user creates a formula with a specific context, such as *AddressBO*, the editor automatically includes *self: AddressBO* in the function signature. QA testers verified that, upon typing `self.` inside the editor, the LSP provided a real-time list of fields defined on the *AddressBO* class (e.g., *AddressLine1, City, PostCode*), as defined in the underlying BOM.

This confirmed that the LSP was correctly interpreting the Python stubs generated from the JSON Schema, and that the editor's connection to the LSP was stable and responsive.

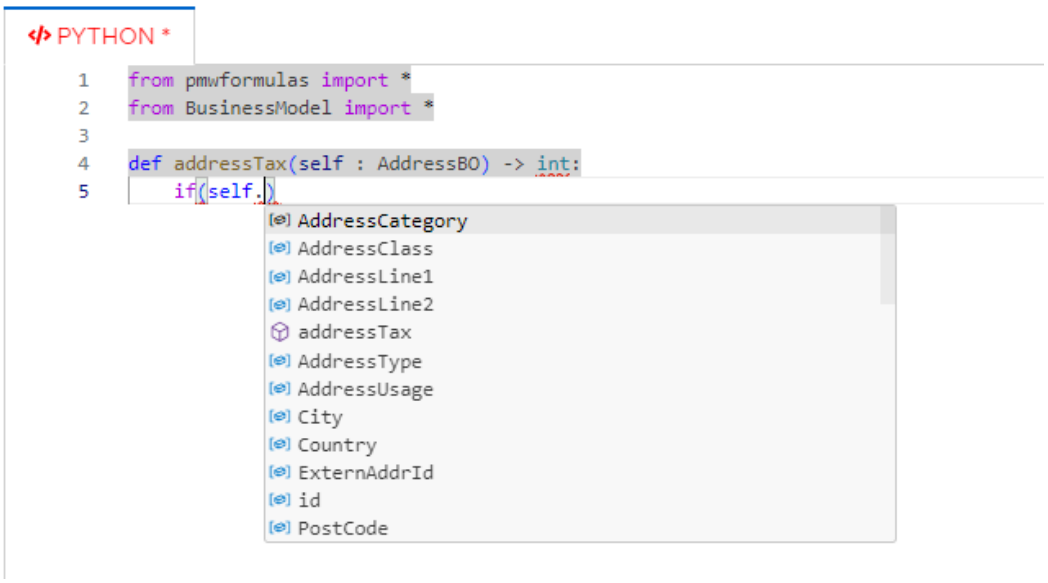



Figure 29 - Autocompletion after typing self. shows context-aware suggestions

### Read-only Enforcement for Imports and Function Headers

To preserve the structure and correctness of each formula, the editor automatically injects required import statements and the function definition header. These lines are essential for static analysis and should not be modified by users. During testing, the QA team attempted to place the cursor within these lines and edit or delete them. The editor successfully blocked all input within those regions and visually marked them as read-only.

This behavior was critical to ensuring that users could not accidentally break the formula's syntax or disrupt the LSP's analysis.

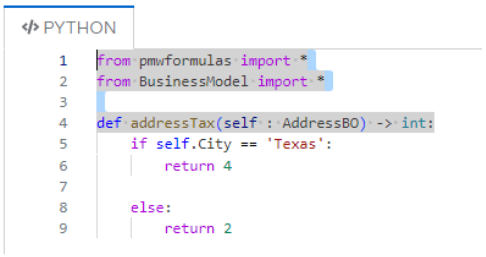



Figure 30 - Read-only header block in the editor, visually marked and locked

## Real-Time Syntax Validation

Another essential feature tested was the system's ability to provide real-time syntax validation. QA testers intentionally wrote both valid and invalid formulas to see whether the LSP correctly flagged issues. For example, when brackets were unclosed, a colon was missing, or an invalid keyword was used, the LSP returned clear diagnostic messages rendered as inline error markers within Monaco Editor.

When the errors were fixed, the messages disappeared automatically, confirming that the document was being synchronized properly with the LSP and revalidated in real time.


Figure 31 - Invalid formula showing red underline and error tooltip returned by Pyright

These QA scenarios helped ensure that the system behaved predictably and supported modeling users in writing accurate and well-structured formulas. In combination with the automated test suite, this validation provided strong assurance that the editor is both technically robust and aligned with real-world usage expectations.

## Enforcement of Maximum Formula Length

To ensure formulas remain within the defined system constraints, the editor enforces a strict limit of 4000 characters per formula. This restriction is essential to maintain compatibility with storage and evaluation logic on the backend.

During testing, the QA team deliberately wrote a formula approaching the character limit, and then attempted to input additional code both by typing and pasting from the clipboard. Once the 4000-character threshold was reached, the editor correctly blocked further input and displayed a message notifying the user that the maximum length had been exceeded.

This behavior plays an important role in preventing data truncation and runtime issues, and it reinforces the editor's ability to guide users toward creating valid and reliable formulas.

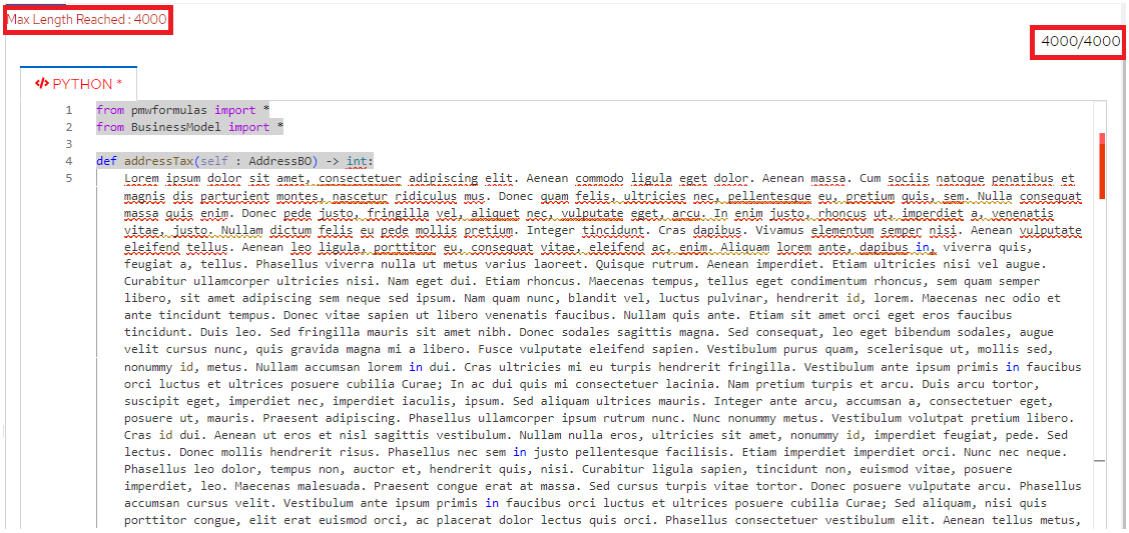



Figure 32 - Editor blocking input after reaching the 4000-character limit and displaying an appropriate warning

# 8 Conclusion

This thesis presented a comprehensive journey through the design and implementation of a formula editing and validation system integrated into PMWeb, a web-based platform tailored for the insurance industry. This work was motivated by a key limitation in PMWeb: the lack of support for behavioral modeling using formulas. These formulas are essential for tasks like validating business rules, calculating premiums, and configuring insurance products.

The project began by identifying the challenges of adding real-time code validation in a web environment. It also looked at how collaborative editing could help make the system more flexible and easier to use. Based on that research, some technologies stood out. Language Server Protocol, that made the finding of Pyright possible, and the Monaco Editor, that brings Pyright functions to light. These tools were selected for their compatibility, extensibility, and strong community support.

The implementation was guided by pragmatic decisions: reusing proven open-source foundations while tailoring them to the specific needs of PMWeb. The creation of a domain-aware LSP, enriched with Business Object Model (BOM) metadata, enabled context-sensitive validation and intelligent autocompletion. Integration with Dolt allowed for version-controlled formulas, promoting collaboration, traceability, and auditability.

We also placed strong emphasis on software quality. From CI/CD pipelines to QA validation and user interface design informed by UX best practices, every layer was considered not just for functionality but for long-term maintainability.

## 8.1 Reflections on Research Question 1

**What are the primary challenges and benefits of implementing real-time syntax-checking, autocomplete suggestions, and validation systems in web-based environments for data-intensive industries?**

Throughout this project, one of the clearest insights was that building real-time intelligent feedback mechanisms in a web-based environment, especially for data-intensive domains like insurance, demands both architectural foresight and careful user experience design. One of the main challenges is balancing the asynchronous behavior of browser-based user interfaces with the strict, synchronous requirements of tools like static analyzers. These two layers don't always align well.

If real-time syntax validation isn't handled carefully, it can lead to issues like delays, race conditions, or incorrect feedback. These problems often happen because the client-side code may not match the current backend state at the moment a validation request is processed.This becomes even more complex in a domain-aware system like PMWeb, where validation must account not only for Python syntax but for business constraints, metadata, and the structure of domain objects. The editor must understand both the language and the problem space.

Achieving accurate autocomplete and feedback required treating the Monaco Editor as more than just a text interface. It had to function as a simulated Python workspace, carefully synchronized with an in-memory language server. This demanded customized virtual file systems, injected context into the editor and intelligent locking of certain code blocks. These strategies enabled the LSP to reason about formulas with full knowledge of the domain by combining an in-memory representation of the user's code with a physical workspace folder containing domain context files. This approach eliminates the need for a traditional runtime environment while still satisfying Pyright’s requirement for a static, file-based structure.

Even though implementing these features was technically challenging, the benefits were clear. Real-time validation makes it easier to catch mistakes early, so users don’t have to rely on trial and error. Autocomplete and tooltips, especially when linked to business object metadata, help users see what types and parameters are available without needing to dig into documentation. This is especially useful in fields like insurance, where precision and traceability are critical.

## 8.2 Reflections on Research Question 2

**How can data model structures be effectively integrated into code editing systems?**

One of the main questions this project explored was how to integrate complex data structures, like JSON-based domain models, into a code editing system. The goal was to make sure that whenever a user writes a Python formula, they can reference the product’s data model directly. This way, the formula logic stays connected to domain definitions, ensuring accuracy and consistency by merging the code editor and business objects together.

The Business Object Model (BOM), defined in JSON Schema, served as the formal expression of this structure. Rather than interpreting it at runtime or embedding it in business logic, it was translated into Python stubs that could be statically analyzed by the LSP. This meant that type validation, autocompletion, and semantic feedback were all aware of the domain's structure, its objects, enums, attributes, and constraints, at every moment of the editing session.

Crucially, this integration respected the multi-version nature of the platform. Each data version had its own BOM, meaning each workspace reflected the precise structural context relevant to the formula being edited. Combined with Dolt's versioned persistence model, this allowed domain logic to evolve safely across branches, supporting experimentation without risking production consistency.

What emerged was a design where the domain model didn't just influence the formula editor, it *shaped* it. Rather than forcing users to learn internal APIs or memorize field names, the editor surfaced this information natively, as part of the coding experience. This tight coupling between domain knowledge and code validation represents a significant step toward more intelligent, user-friendly modeling tools, especially in fields where domain logic is complex and heavily governed, as in insurance.

## 8.3 Future Enhancements

Looking ahead, several enhancements could make the system more powerful and adaptable. One possibility is adding support for other languages, such as OCL, which would extend the system's usefulness for teams that rely on different constraint definitions. Support for dynamic updates to the BOM at runtime is another area of interest, allowing the editor to reflect changes to the data model without needing to restart services. Improving formula reuse through modular imports could also simplify the management of complex logic across projects.

### Dynamic BOM Handling

At present, the *BOM* is static and pre-generated for each data version. Although this ensures consistency, it limits flexibility during active modeling sessions. In the future, the goal is to support **dynamic BOM updates per data version and workspace**, allowing users to modify the BOM structure without requiring a server restart or reinitialization of the LSP container. This would significantly improve the modeling workflow, especially in collaborative or iterative environments.

Implementing such enhancements would involve reloading or regenerating the Python stubs on-the-fly and refreshing Pyright's internal type context during runtime to reflect the updated model. This will enable the BOM to evolve in real time alongside formulas and product configurations.

### Cross-Formula Imports and Reusability

Another key evolution point is enabling **formulas to call each other** naturally using Python-style imports. Since each formula persisted as a physical Python file in the workspace folder tied to a specific data version, this setup can be extended to generate proper module structures and import paths. Doing so will allow named formulas to be reused in other expressions.

By supporting imports between formulas and exposing them through the virtual file system, Pyright will be able to resolve and validate these references just like any standard Python module.

Ultimately, this project enhanced PMWeb's technical foundation and laid the groundwork for a more autonomous, scalable, and intuitive approach to insurance product modeling. It reflects a broader shift in enterprise software: empowering users with tools that are smart, flexible, and built for the web.

## 8.4 Limitations

Even though the solution presented in this thesis went through serious validation with manual and automated tests, no usability testing was done by real end users. The version of the application that has this functionality integrated will be released soon and we will get feedback from a real-world insurance environment to further assess its practical effectiveness.

## 8.5 Final Thoughts

At its core, this project was about more than just formulas, it was about bridging the gap between business logic and developer experience in a way that respects both precision and usability. By fusing web technologies, language tooling, and structured domain models, this thesis shows how modern platforms can empower users to express complex ideas cleanly, safely, and collaboratively. The journey ahead includes richer language support, smarter reuse, and dynamic adaptability, but the foundation laid here proves that intelligent, version-aware, and domain-driven formula modelling is not only possible in the web, but highly effective.

# 9 References


[1] M. J. Page *et al.*, “The PRISMA 2020 statement: an updated guideline for reporting systematic reviews,” *BMJ*, vol. 372, Mar. 2021, doi: 10.1136/BMJ.N71.

[2] R. Saini, S. Bali, and G. Mussbacher, “Towards web collaborative modelling for the user requirements notation using eclipse che and theia IDE,” *Proceedings - 2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering, MiSE 2019*, pp. 15–18, May 2019, doi: 10.1109/MISE.2019.00010.

[3] R. Saini and G. Mussbacher, “Towards Conflict-Free Collaborative Modelling using VS Code Extensions,” *Companion Proceedings - 24th International Conference on Model-Driven Engineering Languages and Systems, MODELS-C 2021*, pp. 35–44, 2021, doi: 10.1109/MODELS-C53483.2021.00013.

[4] H. T. Vu, “Web-based Integrated Development Environment,” *Master’s Projects*, Dec. 2016, doi: https://doi.org/10.31979/etd.f6v3-wqu3.

[5] M. Kazemitabaar, V. Chyhir, D. Weintrop, and T. Grossman, “CodeStruct: Design and Evaluation of an Intermediary Programming Environment for Novices to Transition from Scratch to Python,” *Proceedings of Interaction Design and Children, IDC 2022*, pp. 261–273, Jun. 2022, doi: 10.1145/3501712.3529733/SUPPL_FILE/3529733-CVOR.PDF.

[6] E. Shemon *et al.*, “Meshing, Language Server Protocol, and other User-Oriented MOOSE Framework Improvements to Enhance Reactor Analysis Capabilities and Workflows,” Sep. 2023, doi: 10.2172/2001106.

[7] D. Jäger, “Frontend-only browser-based modeling tools”, doi: 10.34726/HSS.2024.118520.

[8] A. M. McNutt, “No Grammar to Rule Them All: A Survey of JSON-style DSLs for Visualization,” *IEEE Trans Vis Comput Graph*, vol. 29, no. 1, pp. 160–170, Jan. 2023, doi: 10.1109/TVCG.2022.3209460.

[9] A. McNutt and R. Chugh, “Projectional Editors for JSON-Based DSLs,” *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pp. 60–70, 2023, doi: 10.1109/VL-HCC57772.2023.00015.

[10] B. Uyanık and A. Sayar, “Developing Web-Based Process Management with Automatic Code Generation,” *Applied Sciences 2023, Vol. 13, Page 11737*, vol. 13, no. 21, p. 11737, Oct. 2023, doi: 10.3390/APP132111737.

[11] D. Groenewegen, Z. Hemel, and E. Visser, “Separation of concerns and linguistic integration in webDSL,” *IEEE Softw*, vol. 27, no. 5, pp. 31–37, Sep. 2010, doi: 10.1109/MS.2010.92.

[12] P. Freudenstein and M. Nussbaumer, “Constructing advanced web-based dialog components with stakeholders - A DSL approach,” *Proceedings - 8th International Conference on Web Engineering, ICWE 2008*, pp. 38–44, 2008, doi: 10.1109/ICWE.2008.39.

[13] Nadeeshaan. Gunasinghe and Nipuna. Marcus, “Language server protocol and implementation : supporting language-smart editing and programming tools,” p. 239, 2022.

[14] D. Bork and P. Langer, “Language Server Protocol: An Introduction to the Protocol, its Use, and Adoption for Web Modeling Tools,” *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 18, no. 9, pp. 9:1-16, Sep. 2023, doi: 10.18417/EMISA.18.9.

[15] H. Bünder and H. Kuchen, “Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol,” *Communications in Computer and Information Science*, vol. 1161 CCIS, pp. 225–245, 2020, doi: 10.1007/978-3-030-37873-8_10.

[16] R. Rodriguez-Echeverria, M. Wimmer, J. L. C. Izquierdo, and J. Cabot, “Towards a language server protocol infrastructure for graphical modeling,” *Proceedings - 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, pp. 370–380, Oct. 2018, doi: 10.1145/3239372.3239383.

[17] Y. Annik and S. Ander, “Design and Implementation of the Language Server Protocol for the Nickel Language,” 2022, Accessed: Nov. 03, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-323490

[18] H. Bünder, “Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages,” *International Conference on Model-Driven Engineering and Software Development*, pp. 129–140, 2019, doi: 10.5220/0007556301290140.

[19] “Battle: VDMJ User Guide - Google Scholar.” Accessed: Nov. 10, 2024. [Online]. Available: https://scholar.google.com/scholar?cluster=17659956427868329 57&hl=en&oi=scholarr

[20] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, “The overture initiative integrating tools for VDM,” *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 1, pp. 1–6, Jan. 2010, doi: 10.1145/1668862.1668864.

[21] “[Part 2] Connect Monaco React Web Editor with Language Server using WebSocket… How hard can it be? | by Nipuna Marcus | Medium.” Accessed: Dec. 08, 2024. [Online]. Available: https://nipunamarcus.medium.com/part-2-connect-monaco-react-web-editor-with-language-server-using-websocket-how-hard-can-it-be-aa66d93327a6

[22] “What is JSON-RPC? Meaning, Examples, Comparison.” Accessed: Dec. 08, 2024. [Online]. Available: https://www.wallarm.com/what/what-is-json-rpc

[23] “GitHub - microsoft/pyright: Static Type Checker for Python.” Accessed: May 10, 2025. [Online]. Available: https://github.com/microsoft/pyright

[24] “GitHub - erictraut/pyright-playground: Website for testing snippets of code using the pyright type checker.” Accessed: May 10, 2025. [Online]. Available: https://github.com/erictraut/pyright-playground

[25] W. Xu *et al.*, “How Well Static Type Checkers Work with Gradual Typing? A Case Study on Python,” *IEEE International Conference on Program Comprehension*, vol. 2023-May, pp. 242–253, 2023, doi: 10.1109/ICPC58990.2023.00039.

[26] “PEP 484 – Type Hints | peps.python.org.” Accessed: May 10, 2025. [Online]. Available: https://peps.python.org/pep-0484/

[27] “GitHub - openlawlibrary/pygls: A pythonic generic language server.” Accessed: May 10, 2025. [Online]. Available: https://github.com/openlawlibrary/pygls

[28] A. L. Aase, “Improving type safety in Rucio.” Accessed: May 10, 2025. [Online]. Available: https://repository.cern/records/2san1-20j54

[29] “What Do Two Dot and Three Dot Mean for Logs and Diffs? | DoltHub Blog.” Accessed: Dec. 08, 2024. [Online]. Available: https://www.dolthub.com/blog/2022-11-11-two-and-three-dot-diff-and-log/

[30] D. Brahmbhatt *et al.*, “An Open-Source Data Storage and Visualization Platform for Collaborative Qubit Control,” *Sci Rep*, vol. 14, no. 1, p. 22703, Mar. 2024, doi: 10.1038/s41598-024-72584-9.

[31] F. Stenbacka, “Extending a data management system with multiversion indexing, branching, and snapshots,” Jun. 17, 2024. Accessed: Nov. 10, 2024. [Online]. Available: https://aaltodoc.aalto.fi/handle/123456789/129302

[32] “Language Server Protocol (Adding Support for Multiple Language Servers to Monaco Editor) | by Rahulkumarsindhav | DSC DDU | Medium.” Accessed: Jan. 03, 2025. [Online]. Available: https://medium.com/dscddu/language-server-protocol-adding-support-for-multiple-language-servers-to-monaco-editor-a3c35e42a98d

[33] “GitHub - TypeFox/monaco-languageclient: Repo hosts npm packages for monaco-languageclient, vscode-ws-jsonrpc, monaco-editor-wrapper, @typefox/monaco-editor-react and monaco-languageclient-examples.” Accessed: May 10, 2025. [Online]. Available: https://github.com/TypeFox/monaco-languageclient

[34] “monaco-languageclient/packages/examples/src/python at main · TypeFox/monaco-languageclient · GitHub.” Accessed: May 10, 2025. [Online]. Available: https://github.com/TypeFox/monaco-languageclient/tree/main/packages/examples/src/python

# 10 Appendix A – Risk Register

| Risk ID | Description | Cause | Effect | Risk Owner | Probability (1-5) | Impact (1-5) | PI Score | Expected Result, No Action | Risk Response Type | Response description |
|---|---|---|---|---|---|---|---|---|---|---|
| | Description of the risk | Cause of the risk | Effect on the project | Name of person who monitors the risk | Group sourced rough estimate of how likely this is to occur | Rough estimate of how significant the impact of this risk | Probability multiplied by Impact | What will happen if the risk becomes an issue and no action is taken | Decision made by group on how to respond to this risk (see above in blue) | How do you know it is time to put the response into play |
| 1 | Lack of access to sensitive data | Regulatory restrictions | Inability to test real-world scenarios | Oscar Folha | 2 | 4 | 8 | Unable to validate system functionality | Mitigate | Use high-fidelity synthetic datasets |
| 2 | Technology obsolescence | Rapidly evolving frameworks and tools | Loss of compatibility | Oscar Folha | 3 | 3 | 9 | System becomes outdated during development | Mitigate | Regularly update dependencies |
| 3 | Integration challenges | Complexity of BOM and formula editor | Delays in functionality implementation | Oscar Folha | 2 | 5 | 10 | Delayed completion of deliverables | Mitigate | Allocate buffer time for debugging |
| 4 | Stakeholder feedback delays | Limited availability of business users | Iterative adjustments delayed | Oscar Folha | 2 | 3 | 6 | Delayed validation and feedback implementation | Mitigate | Set fixed timelines for user reviews |
| 5 | Performance bottlenecks in real-time validation | Large datasets and formula complexity | Reduced responsiveness | Oscar Folha | 3 | 5 | 15 | Poor system performance during formula validation | Mitigate | Optimize performance with cloud resources and code best practices |