Universität Stuttgart
Institut für Fahrzeugtechnik Stuttgart

Vertraulich

Wie schreibe ich eine Masterarbeit

How do I write a master's thesis

Masterarbeit

von

cand. emob abc def
Matr. Nr. 7654321

Institutsbetreuer:	Mathias Jaksch, M.Sc.
Externer Betreuer:	Dipl.-Ing. Hans Wurst, VW Group
Prüfer:	Prof. Dr.-Ing. H.-C. Reuss

Vorgelegt an der Universität Stuttgart
Institut für Fahrzeugtechnik Stuttgart
Lehrstuhl Kraftfahrzeugmechatronik

2021

Universität Stuttgart
Institut für Fahrzeugtechnik Stuttgart

Lehrstuhl
Kraftfahrzeugmechatronik
Prof. Dr.-Ing. H.-C. Reuss

Tel. (+49) 711 685 – 68501
Fax (+49) 711 685 – 68533

info@ifs.uni-stuttgart.de

31.10.2020 / HCR / JM

Masterarbeit

für

Herrn cand. emob abc def

Matr. Nr. 7654321

Thema: Wie schreibe ich eine Masterarbeit

How do I write a master's thesis

Textblock Zeile 01
Textblock Zeile 02
Textblock Zeile 03
Textblock Zeile 04
Textblock Zeile 05
Textblock Zeile 06
Textblock Zeile 07
Textblock Zeile 08
Textblock Zeile 09
Textblock Zeile 10
Textblock Zeile 11
Textblock Zeile 12
Textblock Zeile 13
Textblock Zeile 14
Textblock Zeile 15
Textblock Zeile 16
Textblock Zeile 17
Textblock Zeile 18
Textblock Zeile 19 – Anmerkung: Dies ist die letztmögliche Zeile auf dieser Seite!

Schutzvermerk:	Die Arbeit ist bis zum 30.04.2024 vertraulich zu behandeln
Institutsbetreuer:	Mathias Jaksch, M.Sc.
Externer Betreuer:	Dipl.-Ing. Hans Wurst, VW Group
Prüfer:	Prof. Dr.-Ing. H.-C. Reuss
Beginn:	31.10.2020
Abgabedatum:	30.04.2021 (Abgabe erfolgte fristgerecht)

Prof. Dr.-Ing. H.-C. Reuss

Erklärung

Hiermit versichere ich, abc def, dass ich die vorliegende Arbeit, bzw. die darin mit meinem Namen gekennzeichneten Anteile, selbständig verfasst und bei der Erstellung der Arbeit die einschlägigen Bestimmungen, insbesondere zum Urheberrechtsschutz fremder Beiträge, eingehalten und nur die angegebenen Quellen und Hilfsmittel benutzt habe.

Soweit meine Arbeit fremde Beiträge (z. B. Bilder, Zeichnungen, Textpassagen) enthält, erkläre ich, dass ich diese Beiträge als solche gekennzeichnet (z. B. Zitat, Quellenangabe) habe und dass ich eventuell erforderliche Zustimmungen der Urheber zur Nutzung dieser Beiträge in meiner Arbeit eingeholt habe.

Für den Fall der Verletzung Rechter Dritter durch meine Arbeit erkläre ich mich bereit, der Universität Stuttgart einen daraus entstehenden Schaden zu ersetzen bzw. die Universität Stuttgart von eventuellen Ansprüchen Dritter freizustellen.

Die Arbeit ist weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen. Ferner ist sie weder vollständig noch in Teilen bereits veröffentlicht worden. Das elektronische Exemplar stimmt mit den anderen Exemplaren überein.

Stuttgart, den 30.04.2021

____

abc def

Preamble

Contents

Acronyms

Symbols

Symbol	Unit	Description
F_{N}	N	Force
τ	Nm	Torque

List of Figures

List of Tables

Kurzfassung

Abstract

1 Introduction

Modern commercial vehicles represent a quintessential example of cyber-physical systems, where sophisticated software enables precise control over complex mechanical components. The software controlling these vehicles has grown exponentially in complexity over recent decades, evolving from simple engine management to comprehensive control of virtually all vehicle functions. At the core of this evolution is the Electronic Control Unit (ECU)—a specialized computer that executes software to manage specific vehicle functions [6]. Contemporary commercial vehicles contain dozens of interconnected ECUs working in concert to ensure optimal performance, efficiency, and safety across diverse operating conditions.

1.1 Background and Context

The automotive industry has undergone a profound transformation over the past decades, evolving from predominantly mechanical systems to highly sophisticated mechatronic platforms [27]. This evolution has been particularly pronounced in the commercial vehicle sector, where modern trucks rely on complex networks of Electronic Control Units to manage everything from engine performance to safety systems [6]. These systems must adapt to a wide range of operational conditions, regulatory requirements, and market-specific configurations, creating a significant challenge in managing software variability.

At the heart of this variability management lies the Common Powertrain Controller (CPC)—a central ECU managing critical functions related to engine and transmission control. The CPC's operation is governed by thousands of configurable parameters that determine how the powertrain behaves under specific conditions [35]. These parameters influence everything from basic engine timing to sophisticated emission control strategies, making their precise configuration essential for vehicle performance, efficiency, and regulatory compliance.

The parameter management challenge is further complicated by the global nature of modern vehicle development. Commercial vehicles must conform to different emissions regulations, operate in diverse environmental conditions, and meet varying customer expectations across global markets. Consequently, a single vehicle model may require numerous parameter configurations, each tailored to specific combinations of market requirements, hardware configurations, and customer specifications [38].

1.2 Problem Statement

The current approach to parameter management in commercial vehicle development relies predominantly on distributed Excel spreadsheets, a methodology that emerged during a period when parameter counts were manageable and development teams were smaller [38]. However, as software complexity has increased exponentially, this fragmented approach has introduced significant limitations and risks to the development process.

Development teams distributed across different locations must coordinate changes to thousands of parameters, track their versions, and ensure consistency across multiple vehicle platforms. The absence of a centralized version control system makes it exceptionally difficult to track changes effectively and manage releases. This situation becomes particularly critical when dealing with safety-critical parameters that directly influence vehicle performance and regulatory compliance.

The manual nature of current processes, combined with the lack of automated validation mechanisms, introduces substantial risks of data inconsistency, version conflicts, and delayed implementation of critical parameter updates. Parameter changes are not consistently verified against established rules and constraints, potentially leading to incompatible configurations or non-compliant behavior [35].

Integration with critical enterprise systems presents another significant challenge. The current process of synchronizing data with internal database systems involves several manual steps, consuming valuable development resources and introducing potential points of failure in the configuration management workflow. The absence of automated data validation and synchronization mechanisms creates additional risks for data integrity and consistency across these interconnected systems.

Furthermore, the increasing emphasis on rapid development cycles and continuous integration in the automotive industry demands a more sophisticated approach to parameter management [6]. The existing system's limitations become particularly apparent when considering the need for simultaneous development of multiple vehicle variants, each requiring specific parameter configurations for different markets and regulatory environments.

These challenges collectively underscore the urgent need for a modern, database-driven solution that can address the complexities of contemporary automotive software development while providing a scalable foundation for future growth and adaptation.

1.3 Research Objectives

This thesis aims to address the fundamental challenges in automotive parameter management through the development of database architecture for VMAP (Variant Management and Parametrization), a web-based application for powertrain parameter configuration. The research objectives encompass both theoretical foundations and practical implementation considerations, focusing on creating a robust solution that meets the complex demands of modern vehicle development processes.

The primary research objective centers on developing a centralized database architecture that can effectively manage the complexity of powertrain parameters while maintaining data integrity and traceability [39]. This architecture must support sophisticated version control mechanisms that can handle parameter variations across different development stages and vehicle variants. The system should provide comprehensive audit trails and change history, enabling development teams to track modifications and understand the evolution of parameter configurations over time.

A second crucial objective focuses on the implementation of a sophisticated version control system that addresses the unique requirements of parameter management in automotive software development. This system must go beyond traditional source code version control approaches to handle the complex relationships between parameters, their variants, and their applications across different vehicle platforms [35]. The version control mechanism should support parallel development streams while maintaining consistency and preventing conflicts in parameter configurations.

The research also aims to establish a comprehensive role-based access control system that supports the diverse needs of different user groups within the development process. This includes creating specialized interfaces and permissions for Module Developers, Documentation Team members, Administrators, and Read-only Users, each with specific capabilities and restrictions aligned with their responsibilities [30]. The access control system must balance security requirements with the need for efficient collaboration among development teams.

Integration with existing enterprise systems represents another critical objective of this research. The VMAP system must establish seamless data exchange mechanisms with internal database systems, ensuring consistent information flow while minimizing manual intervention [6]. This integration should support automated validation of parameter changes and provide mechanisms for maintaining data consistency across different systems.

A final key objective involves the development of database interfaces and query optimization strategies that will support the web-based interface implementation. While the actual User Interface (UI) development falls outside the scope of the thesis, the research will focus on designing efficient database structures, stored procedures, and APIs that enable seamless integration with the planned web interface [27]. This includes developing optimized query patterns for complex operations such as parameter comparison, variant management, and release workflows, while ensuring robust data validation and business rule enforcement.

1.4 Significance of the Study

The significance of this research extends beyond addressing immediate technical challenges in parameter management. By developing a comprehensive database solution for variant management and parametrization, this work contributes to the broader field of automotive software engineering in several important ways.

First, the research advances the understanding of version control in parameter-centric systems, extending traditional concepts of software versioning to accommodate the unique characteristics of automotive parameter configurations. While considerable research has been conducted on code versioning, the versioning of parameter data presents distinct challenges that require specialized approaches [3]. This thesis contributes to closing this gap by developing and evaluating new methods for parameter versioning in complex automotive systems.

Second, the work addresses critical industry needs for improved quality and efficiency in vehicle development. Commercial vehicle manufacturers face increasing pressure to reduce development time while managing growing software complexity and ensuring regulatory compliance across global markets [6]. By providing a more robust and efficient parameter management solution, this research directly contributes to these industry priorities, potentially reducing development costs and improving vehicle quality through more consistent parameter configurations.

Third, the research advances the integration of database technology with domain-specific engineering processes. By developing specialized database structures and functions tailored to the unique requirements of automotive parameter management, this work demonstrates how database technology can be adapted to support complex engineering workflows [11]. This integration perspective is valuable not only for automotive applications but also for other engineering domains facing similar challenges in managing complex, highly variable system configurations.

Finally, the research contributes to the growing field of model-based systems engineering by providing a structured approach to managing the parametric aspects of system models. As the automotive industry continues to adopt model-based approaches for system development, the management of parameter configurations becomes increasingly critical for maintaining model integrity and traceability [35]. This thesis provides insights and solutions that support this evolution toward more systematic model-based development practices.

1.5 Thesis Structure

The thesis is organized into six chapters that systematically address the research objectives and present a comprehensive solution for automotive parameter management. The structure follows a logical progression from theoretical foundations through practical implementation, ensuring thorough coverage of both academic and industry perspectives.

Following this introduction, Chapter 2 presents a comprehensive review of the state of the art in database version control systems and automotive parameter management. This chapter examines existing approaches to software configuration management in the automotive industry [27], analyzes current database versioning techniques [3], and evaluates their applicability to parameter management systems. The review encompasses both academic research and industry practices, providing a solid foundation for the proposed solution.

Chapter 3 details the methodology and concept development, beginning with a thorough requirements analysis based on industry needs and academic best practices [35]. This chapter explores the system architecture design, consisting of the database schema, version control mechanisms, and user management frameworks. Particular attention is given to the integration requirements with existing systems and the development of robust validation mechanisms for parameter management.

The implementation strategy and technical design are presented in Chapter 4, which outlines the practical realization of the VMAP system. This chapter describes the development of the database structure, the implementation of version control mechanisms, and the creation of the database interfaces. The chapter also details the integration approaches with internal database systems, highlighting the technical challenges and solutions developed during the implementation phase [6].

Chapter 5 focuses on system evaluation and validation, presenting a comprehensive assessment of the VMAP system against the defined research objectives. This chapter includes detailed performance analyses, user acceptance testing results, and comparative evaluations against existing parameter management solutions. The evaluation framework incorporates both quantitative metrics and qualitative assessments to provide a thorough understanding of the system's effectiveness [11].

The thesis concludes with Chapter 6, which summarizes the research findings and presents recommendations for future development. This chapter reflects on the contributions of the research to both academic knowledge and industry practice, discussing the implications for automotive software development and configuration management. Additionally, it outlines potential areas for future research and system enhancement based on the insights gained during the project.

Throughout these chapters, the research methodology combines theoretical analysis with practical implementation, ensuring that the resulting system meets both academic standards and industry requirements. Special attention is given to database versioning approaches, user role management, and integration strategies with existing systems, addressing the unique challenges of automotive software configuration management [35].

1.6 Project Plan

The research project follows a structured approach spanning six months from November 2024 to April 2025, organized into three distinct phases: Exposé, Implementation, and Finalization. The comprehensive timeline ensures systematic progression through all research objectives while maintaining academic rigor and quality standards.

1.6.1 Exposé Phase (November - January)

The initial phase focuses on establishing strong theoretical foundations and gathering comprehensive requirements. Literature review constitutes a significant portion of this phase, extending over six weeks to ensure thorough coverage of current database versioning approaches, parameter management systems, and industry practices in automotive applications. This review encompasses analysis of existing version control systems, examination of industry standards for software configuration management, and evaluation of current parameter management solutions.

Requirements analysis follows the literature review, spanning three weeks to capture detailed system specifications. This phase involves extensive stakeholder consultation to document system requirements, analyze existing Excel-based workflows, define integration requirements with internal database systems, and establish user roles and access control specifications. The Exposé phase concludes with the submission of a comprehensive research proposal at the end of Week 3 in January.

1.6.2 Implementation Phase (December - March)

The implementation phase encompasses four major components, each allocated four weeks for development and refinement. Database design initiates this phase, focusing on developing the schema for parameter management, designing version control mechanisms, creating data models for user management, and planning integration interfaces with existing systems.

System architecture development follows, concentrating on overall system design, version control workflows, user management frameworks, and validation mechanisms. This stage establishes the foundational structure for the entire system while ensuring alignment with identified requirements and industry standards.

Prototype development constitutes the third component, involving implementation of core database functionality, development of version control features, creation of user management interfaces, and construction of system integration components. This stage transforms theoretical designs into practical implementations while maintaining focus on system usability and performance.

The final component of this phase involves comprehensive testing and validation, including database performance testing, validation of version control mechanisms, testing of user management functions, and verification of system integration capabilities. This stage ensures all implemented features meet specified requirements and performance standards.

1.6.3 Finalization Phase (April)

The concluding phase focuses on documentation and thesis preparation over four weeks. The first two weeks are dedicated to comprehensive documentation, including compilation of implementation details and system architecture documentation.

The subsequent two weeks concentrate on thesis writing, involving comprehensive documentation of research findings, inclusion of test results and analysis, preparation of conclusions and recommendations, and thorough content review and refinement. The phase concludes with thesis submission in Week 16 and final project presentation in Week 17.

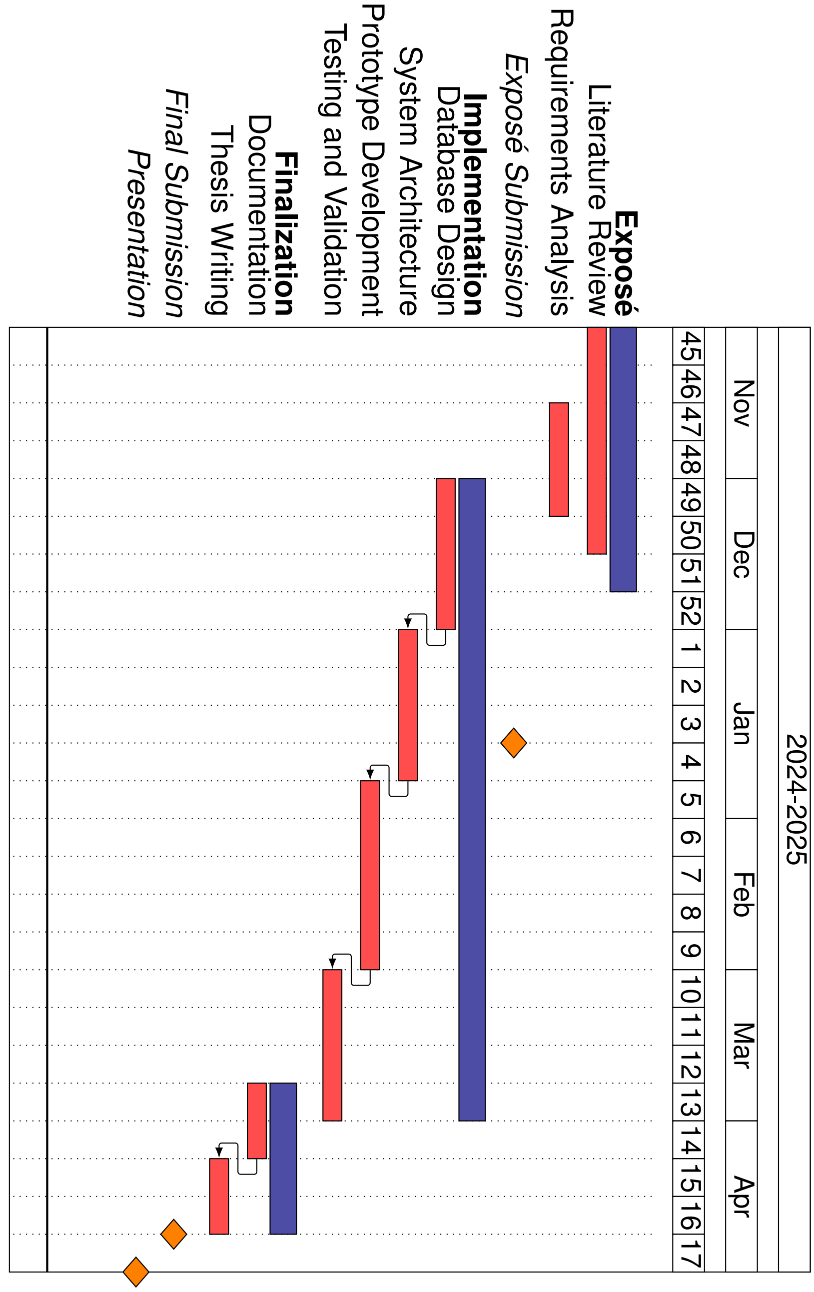


Figure 1.1: Gantt Chart of the planned work schedule.

2 Theoretical Background

This chapter establishes the theoretical foundation necessary for understanding the design and implementation of the Variant Management and Parametrization (VMAP) database system. It begins with an overview of automotive electronic control systems and parameter management, explaining the fundamental concepts that drive the requirements for the VMAP system. The chapter then explores database management systems, database design methodologies, and access control models relevant to the implementation. Finally, it discusses version control concepts and temporal database management approaches, which are critical for the parameter versioning requirements in automotive software development.

2.1 Automotive Electronic Control Systems

Modern commercial vehicles contain dozens of Electronic Control Units (ECUs) that manage various vehicle subsystems. Each ECU is a specialized computing device that controls specific functions through software parameters [35]. Understanding the structure and organization of these systems is essential for designing an effective parameter management solution.

2.1.1 ECU Hierarchy and Parameter Organization

Automotive electronic systems follow a hierarchical organization that structures parameters into logical groupings. At the top level, Electronic Control Units (ECUs) represent distinct hardware components controlling specific vehicle functions such as engine management, transmission control, or brake systems [19]. Within each ECU, modules represent functional software units that implement specific capabilities such as cruise control, adaptive power steering, diagnosis. Each module contains Parameter IDs (PIDs) that group related parameters, and finally, individual parameters define specific configuration values that affect system behavior [36].

This hierarchical structure is not merely organizational but reflects the actual architecture of automotive electronic systems, where software components are modularized for maintainability, reusability, and functional separation. The Common Powertrain Controller (CPC), a central ECU in modern trucks, manages critical powertrain functions

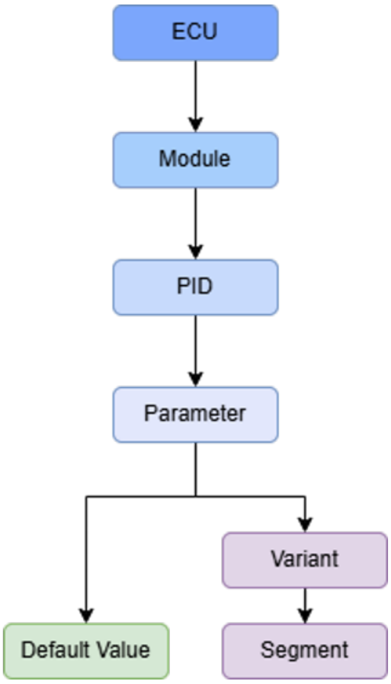


Figure 2.1: Hierarchical Organization of Automotive Electronic Systems

through thousands of configurable parameters organized into this hierarchical structure [35].

Parameters themselves have complex characteristics beyond simple values. They can be scalar values, one-dimensional arrays (curves), or multi-dimensional arrays (maps or tables). Each parameter has specific attributes defining its data type, valid range, engineering units, scale factors, and default values [27]. For instance, an engine timing map might be represented as a two-dimensional array where engine speed and load are the independent variables, and ignition timing angle is the dependent variable. This complexity in parameter structure creates specific requirements for the database system designed to manage them.

2.1.2 Parameter Variants and Customization

A fundamental challenge in automotive parameter management is supporting multiple parameter configurations for different vehicle variants, regional requirements, and operating conditions. Rather than maintaining separate complete parameter sets for each configuration, which would lead to significant redundancy, automotive systems implement a variant mechanism that allows selective overriding of parameter values based on specific conditions [35].

In this approach, each parameter has a default value defined in the baseline configuration. Variants are created to represent specific vehicle configurations or conditions, and segments define modified parameter values within these variants. If no segment exists for a particular parameter in an applicable variant, the default value is used. This approach minimizes redundancy by storing only the modified values rather than complete parameter sets for each configuration [6].

Figure 2.2: Parameter Variant and Segment Concept

Variants are associated with code rules—boolean expressions that determine when a variant applies based on vehicle configuration codes. For example, a variant might apply only to vehicles with a specific engine type and transmission combination, or to vehicles destined for a particular market with unique regulatory requirements. The code rule evaluation process selects the appropriate variants for a specific vehicle configuration during parameter file generation [36].

This variant approach creates specific requirements for the database system, which must efficiently store and retrieve variant definitions and segment values while maintaining the relationships between parameters, variants, and segments. The system must also implement a parameter resolution process that correctly applies variants based on vehicle configuration codes, ensuring that the right parameter values are used for each specific vehicle.

2.1.3 Release and Phase Management

Automotive software development follows a structured release process with well-defined phases representing increasing levels of maturity and stability [6]. For parameter management, this translates into a phase-based development process where parameter configurations evolve through sequential stages before being released for production.

The typical release cycle in automotive parameter development consists of bi-annual releases (e.g., "24.1" and "24.3" for first and third quarters of 2024), with each release progressing through four sequential phases: Initial, PreTest1, PreTest2, and Final [27]. Different ECUs may progress through these phases at different rates, requiring the parameter management system to support concurrent work on multiple phases.

Each phase represents a milestone in the development process with specific activities and quality gates. The Initial phase involves the creation of new parameters and initial configuration. PreTest1 and PreTest2 phases involve refinement based on testing

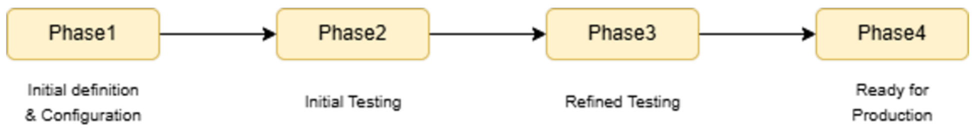


Figure 2.3: Automotive Parameter Release Cycle

feedback, with increasing levels of validation. The Final phase represents the completed configuration ready for production release [35].

When a phase transitions to the next stage, parameter configurations are copied forward, establishing a new baseline for continued development. Changes made in earlier phases should propagate to later phases unless explicitly overridden, creating a complex versioning requirement for the parameter management system [27]. Additionally, at specific development milestones, phases may be "frozen" to create stable reference points for documentation and testing, requiring the parameter management system to enforce read-only access to frozen phases while still allowing continued development in active phases.

This phase-based release process establishes specific requirements for the database system's versioning model, which must maintain distinct parameter configurations for each phase while supporting phase transitions, change propagation, and selective freezing. The versioning approach must align with this development process rather than implementing a generic temporal model, ensuring that the system supports the actual workflows used in automotive parameter development.

2.2 Database Management Systems

Database management systems (DBMS) serve as the foundation for structured information storage and retrieval. They provide mechanisms for storing, organizing, and accessing data while ensuring integrity, security, and concurrent access [11]. For the VMAP system, selecting an appropriate database approach is critical for meeting the complex requirements of automotive parameter management.

2.2.1 Relational Database Management Systems

Relational Database Management Systems (RDBMS) organize data into structured tables composed of rows and columns, based on the relational model proposed by E.F. Codd in 1970 [8]. The relational model establishes a mathematical foundation for representing data as relations (tables) with well-defined operations for data manipulation. This approach has dominated database technology for decades due to its solid theoretical foundation and practical advantages for structured data management.

In relational databases, tables adhere to predefined schemas that specify the structure, data types, and constraints applicable to the data. Each table typically includes a primary key that uniquely identifies each row, while foreign keys establish relationships between tables, implementing the referential integrity that ensures consistency across related data [11].

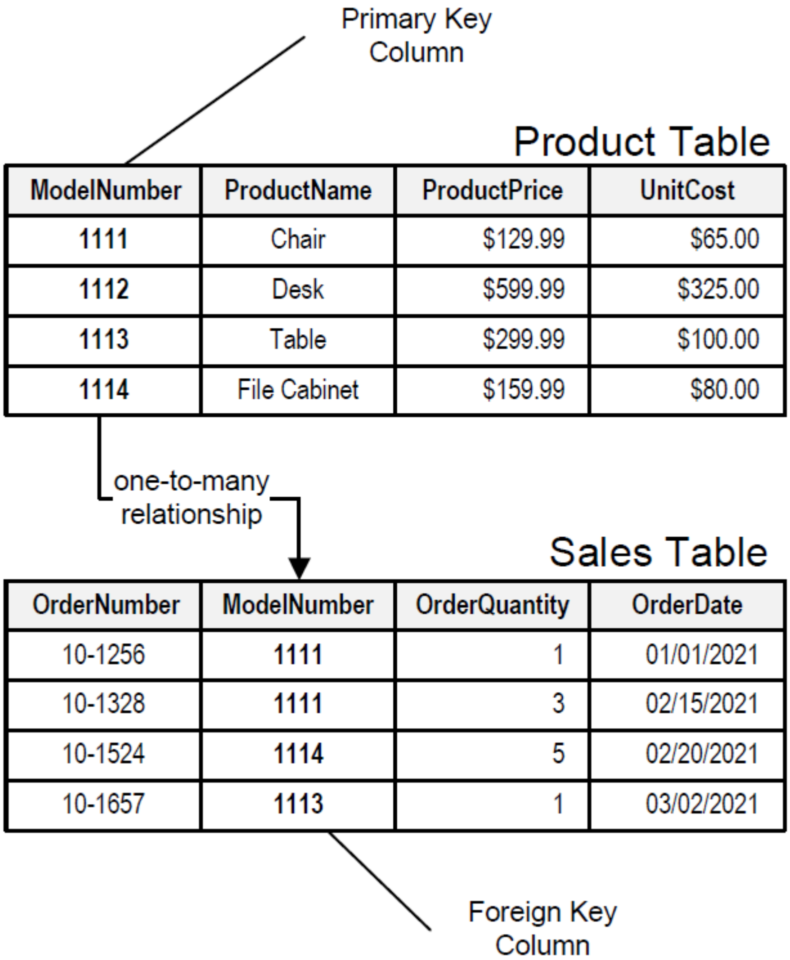
A key strength of relational databases is their adherence to ACID properties (Atomicity, Consistency, Isolation, Durability), which ensure reliable transaction processing. Atomicity guarantees that transactions are treated as indivisible units that either complete entirely or have no effect. Consistency ensures that transactions maintain database integrity by transforming the database from one valid state to another. Isolation prevents interference between concurrent transactions, making them appear as if executed sequentially. Durability ensures that committed transactions persist even after system failures [11].

ACID compliance makes relational databases particularly suitable for automotive parameter management, where data integrity and consistency are paramount. Incorrect parameter values could potentially affect vehicle safety and performance, making the strong consistency guarantees of relational databases essential for maintaining data integrity [35]. Additionally, the hierarchical structure of automotive parameter systems—with well-defined relationships between ECUs, modules, PIDs, and parameters—aligns naturally with the relational model's representation of structured data and relationships.

2.2.2 Non-Relational Database Systems

Non-relational databases, often referred to as NoSQL (Not Only SQL) databases, emerged as alternatives to the relational model, particularly for use cases involving large-scale distributed systems, unstructured data, or schema flexibility requirements. Unlike relational databases, NoSQL systems typically sacrifice some aspects of ACID compliance in favor of scalability, flexibility, and performance characteristics suited to specific application domains [3].

NoSQL databases can be categorized into several types based on their data models: document databases (MongoDB, CouchDB), key-value stores (Redis, DynamoDB), column-family stores (Cassandra, HBase), and graph databases (Neo4j, Amazon Neptune). Many NoSQL systems follow the BASE principle (Basically Available, Soft



Product Table

ModelNumber	ProductName	ProductPrice	UnitCost
1111	Chair	$129.99	$65.00
1112	Desk	$599.99	$325.00
1113	Table	$299.99	$100.00
1114	File Cabinet	$159.99	$80.00

Sales Table

OrderNumber	ModelNumber	OrderQuantity	OrderDate
10-1256	**1111**	1	01/01/2021
10-1328	**1111**	3	02/15/2021
10-1524	**1114**	5	02/20/2021
10-1657	**1113**	1	03/02/2021

Figure 2.4: Example of a Relational Schema [37]

state, Eventually consistent) rather than ACID, prioritizing availability and partition tolerance over immediate consistency [**?**].

While NoSQL databases excel in specific domains such as high-volume web applications, real-time analytics, and social networks, they present challenges for applications requiring complex transactions, strict data integrity, or sophisticated query capabilities across related entities [20]. For automotive parameter management, these limitations make NoSQL systems generally less suitable than relational databases.

The potential for eventual consistency rather than immediate consistency in many NoSQL systems could lead to incorrect parameter configurations being used during development or testing, creating significant risks for vehicle performance and safety. Additionally, the hierarchical nature of automotive electronic systems, with well-defined relationships between entities, aligns naturally with the relational model's approach to representing structured data and relationships. The ability to enforce these relationships through foreign key constraints provides important safeguards against data inconsistency that would be more difficult to implement in many NoSQL systems.

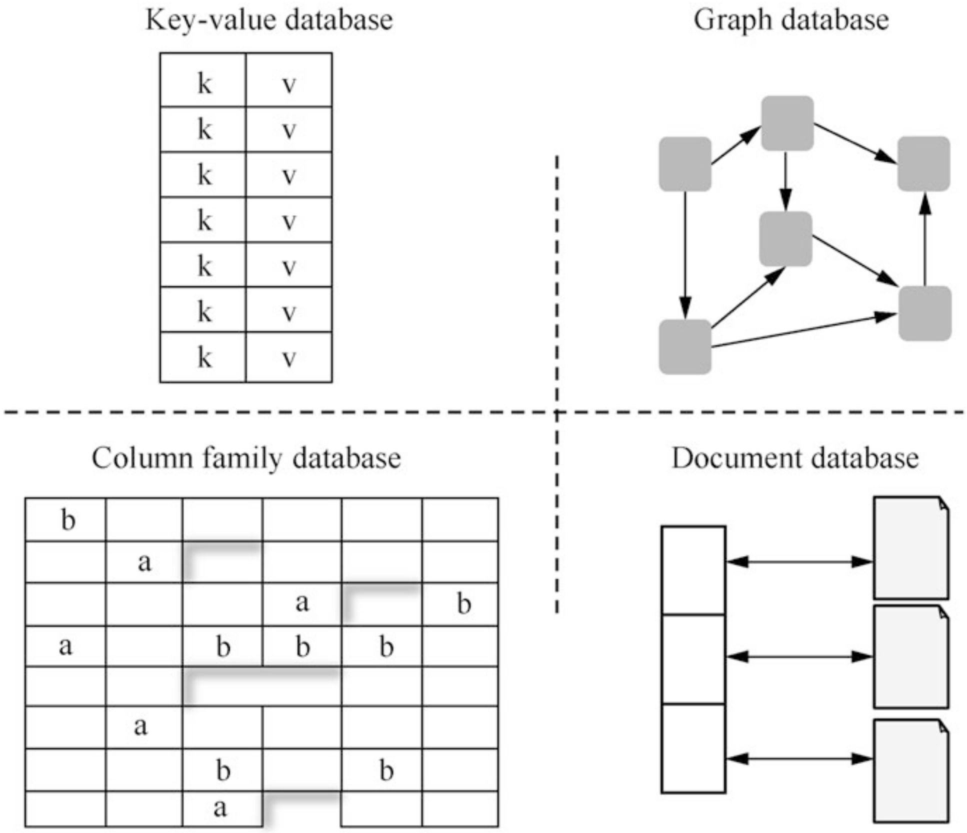


Figure 2.5: Major Types of NoSQL Databases [14]

2.3 Database Design Methodologies

Database design methodologies provide structured approaches to creating efficient, reliable database systems. These methodologies help translate real-world information needs into technical implementations that can store and manage data effectively. This section explores fundamental approaches that form the theoretical foundation for database design, presented in a sequence that follows the natural progression from user requirements to technical implementation.

2.3.1 Use Case Modeling

Before designing a database structure, it is essential to understand how users will interact with the system. Use case modeling provides a technique for capturing user requirements by identifying who will use the system (actors) and what they need to accomplish (use cases). Developed by Ivar Jacobson, use case modeling has become a cornerstone of requirements analysis in system development [17].

A use case represents a specific goal that an actor wishes to achieve using the system. Actors can be human users with different roles (such as administrators or regular users) or external systems that interact with the database. The collection of all use cases defines the system's functional boundaries—what it must do to satisfy user needs [17].

Use case diagrams provide a visual representation of these relationships, showing actors as stick figures and use cases as ovals, with lines connecting actors to their associated use cases. This visual format makes the system's purpose accessible to non-technical stakeholders, facilitating communication between developers and users. As noted by Jacobson, "Use cases bridge the gap between the users' and the developers' views of the system" [17].

For automotive parameter management systems, use case modeling helps identify the different ways in which engineers, documentation specialists, administrators, and other stakeholders need to interact with parameter data. These use cases then inform the database design, ensuring that the resulting structure effectively supports all required operations.

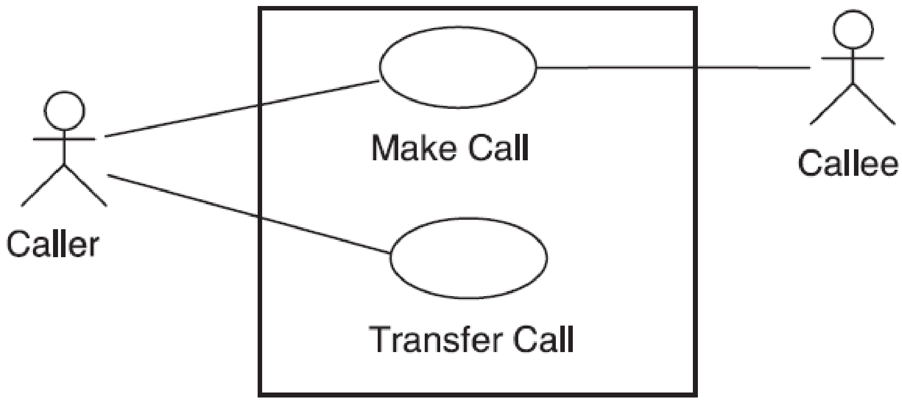


Figure 2.6: Use Case Diagram of switching system [17]

2.3.2 Entity-Relationship Modeling

After understanding user requirements through use cases, the next step is to model the data itself. Entity-Relationship (ER) modeling provides a conceptual framework for representing the data structure needed to support the identified use cases. Introduced by Peter Chen in 1976, ER modeling has become the most widely used approach for conceptual database design [7].

ER modeling identifies three main components:

Entities represent the objects or concepts about which information needs to be stored. In an automotive context, these might include vehicles, electronic control units (ECUs), parameters, and users. Entities are represented as rectangles in ER diagrams.

Attributes describe the specific properties or characteristics of each entity. For example, a parameter entity might have attributes like name, value, unit, and description. Attributes are shown as ovals connected to their entity.

Relationships describe the associations between entities. For instance, "ECUs contain parameters" expresses a relationship between ECU and parameter entities. Relationships are shown as diamonds connecting the related entities, with cardinality notations indicating how many instances of each entity can participate in the relationship [11].

ER modeling is particularly valuable for complex domains like automotive systems because it provides a visual representation that stakeholders can understand while being precise enough to guide database implementation. Chen explains that "the entity-relationship model adopts the more natural view that the real world consists of entities and relationships" [7], making it an intuitive approach for modeling real-world systems.

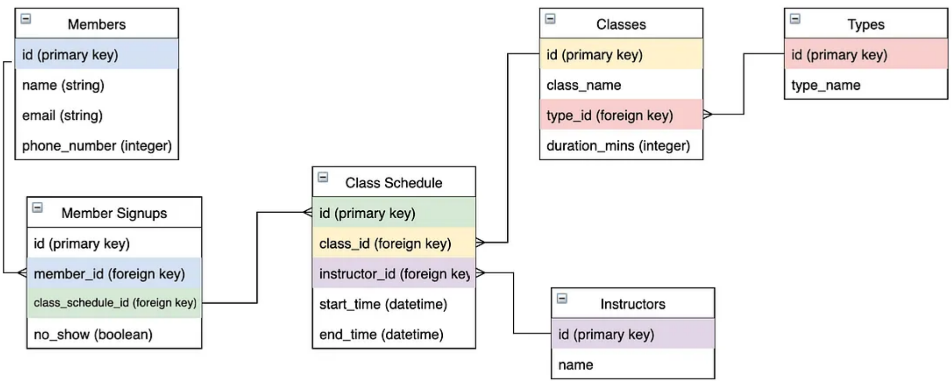


Figure 2.7: Entity-Relationship Diagram for Core VMAP Entities [23]

2.3.3 Database Normalization

Once the conceptual model is established through ER modeling, database normalization helps refine this model into an efficient, consistent structure. Normalization is a systematic process developed by E.F. Codd that organizes data to minimize redundancy and avoid update anomalies [8].

Normalization proceeds through several "normal forms," each addressing specific types of data inconsistencies:

First Normal Form (1NF) requires that each cell in a table contains only a single value, not a list of values. For example, storing multiple phone numbers in a single field would violate 1NF. This ensures that data is atomic (indivisible) and can be manipulated consistently.

Second Normal Form (2NF) builds on 1NF by requiring that all non-key attributes depend on the entire primary key, not just part of it. This prevents situations where changing one piece of data requires multiple updates in different places.

Third Normal Form (3NF) further refines the structure by requiring that non-key attributes depend only on the primary key, not on other non-key attributes. This eliminates transitive dependencies that can lead to update anomalies [11].

For most practical applications, achieving 3NF provides a good balance between data integrity and system performance. As explained by Date, "Third normal form is considered adequate for most practical purposes; further normalization is usually performed only when necessary" [10].

In automotive parameter management, normalization helps organize complex data about ECUs, modules, and parameters into a structure that maintains consistency while supporting efficient access. For example, normalizing parameter data ensures that when a parameter value changes, that change only needs to be recorded in one place, eliminating the risk of inconsistent values across the database.

2.3.4 Role-Based Access Control Models

Database systems often contain sensitive information that should not be accessible to all users. Role-Based Access Control (RBAC) provides a structured approach to managing permissions within a database system. Introduced by David Ferraiolo and Richard Kuhn in the 1990s, RBAC has become the predominant model for access control in enterprise systems due to its balance of security and administrative simplicity [30].

The core concept of RBAC is that permissions are associated with roles, and users are assigned to appropriate roles rather than being granted permissions directly. A role represents a specific function within an organization, such as “administrator,” “engineer,” or “analyst.” Each role is granted a set of permissions that allow users assigned to that role to perform specific operations on database objects like reading, creating, updating, or deleting records [12].

This structure provides several important advantages over direct permission assignment. First, it simplifies administration by allowing permissions to be managed at the role level rather than the individual user level. When a new user joins the organization, they can simply be assigned to the appropriate roles rather than requiring configuration of individual permissions. Second, it improves security by implementing the principle of least privilege, ensuring that users have only the permissions necessary for their specific responsibilities which reduces the risk of unauthorized access or accidental data modifications [30].

The theoretical foundation of RBAC includes several key components: users (individuals who need access to the system), roles (collections of permissions that correspond to job functions), permissions (defined operations on specific resources), and sessions (temporary bindings between users and their assigned roles). These components provide a flexible framework for implementing access control policies tailored to specific organizational needs while maintaining a clear separation between users and permissions through the role abstraction [29].

2.3.5 Version Control for Databases

Version control for databases addresses the challenge of tracking changes to data structures and content over time. Unlike traditional file-based version control systems designed for source code, database versioning must maintain complex relationships between entities while preserving historical states and supporting evolution through distinct development stages [3].

Several approaches have emerged for implementing version control in database systems. The snapshot approach captures complete database states at specific points in time, providing simple retrieval of historical states but potentially consuming significant storage resources. The change-based approach records only modifications to database content, reducing storage requirements but requiring reconstruction of historical states through the application of change records [3].

The temporal approach extends traditional database structures with time dimensions, enabling direct querying of historical states through time-based predicates. This approach typically introduces valid time (when facts are true in the real world) and transaction time (when facts are recorded in the database) dimensions, allowing sophisticated historical analysis but adding complexity to schema design and query formulation [33].

Phase-based versioning represents a domain-specific approach that aligns database versioning with development phases rather than continuous time. This approach explicitly models development stages as first-class entities in the database schema, associating data with specific phases rather than temporal timestamps. According to Bhattacherjee et al., "Domain-specific versioning approaches often provide better performance and usability than generic temporal database techniques when tailored to specific application requirements" [3].

2.3.6 Temporal Database Concepts

Many applications, including automotive parameter management, need to track how data changes over time. Temporal database concepts address these requirements by providing mechanisms for managing time-varying data. Unlike traditional databases that store only the current state, temporal databases maintain historical states and support queries based on time dimensions [33].

Temporal databases typically support two key time dimensions. Valid time represents when facts are true in the modeled reality—for example, when a particular parameter configuration becomes active in a vehicle. Transaction time represents when facts are recorded in the database—for example, when a parameter value was updated in the system. Databases that support both dimensions are known as bi-temporal databases [22].

Temporal database implementations often use specialized table structures called temporal tables. These tables extend traditional table structures with additional timestamp columns that define the time periods during which each record is valid. For example, a temporal parameter table might include ValidFrom and ValidTo columns that define when each parameter value is applicable, allowing the database to maintain a complete history of parameter changes [28].

Kulkarni and Michels explain that "temporal tables provide a systematic way to track and query historical data without requiring application-level version management" [22]. This capability is particularly valuable in regulated industries like automotive development, where traceability and auditability of parameter changes are essential for compliance and quality assurance.

In automotive parameter management, temporal database concepts can support critical requirements such as tracking parameter evolution throughout the development lifecycle, maintaining historical records for diagnostic and compliance purposes, and enabling historical analysis to understand how parameter configurations have evolved over time. These capabilities form an important theoretical foundation for designing systems that manage time-sensitive data in complex domains.

ENo	EStart	EEnd	EDept
22217	2010-01-01	2011-02-03	3
22217	2011-02-03	2011-09-10	4
22217	2011-09-10	2011-11-12	3

Figure 2.8: Temporal Database Example [22]

2.3.7 Strategic Denormalization

While normalization provides a theoretical foundation for database integrity, practical database design often requires balancing normalization principles with performance considerations. Strategic denormalization involves deliberately introducing controlled redundancy to improve performance for specific operations [3].

Consider a fully normalized database where information about parameters, their modules, and their ECUs is stored in separate tables. To retrieve a parameter with its associated module and ECU information would require joining all three tables—an operation that becomes increasingly expensive as the database grows. In cases where this retrieval happens frequently, storing the module and ECU names directly in the parameter table (introducing controlled redundancy) could significantly improve performance [31].

Molinaro emphasizes that "denormalization is not about abandoning normalization principles, but about making strategic exceptions for performance reasons" [24]. These exceptions should be carefully documented and justified based on specific performance requirements.

For automotive systems, where both data integrity and query performance are critical, finding the right balance between normalization and strategic denormalization is essential. This balance ensures that parameter data maintains consistency while providing the performance needed for engineering workflows.

2.3.8 Conceptual, Logical, and Physical Design Levels

Database design typically proceeds through three levels of abstraction, allowing designers to manage complexity by focusing on different aspects at each stage:

Conceptual design focuses on what data needs to be stored, without concern for implementation details. The ER model created at this stage captures entities, attributes, and relationships from a business perspective, providing a foundation that both technical and non-technical stakeholders can understand [11].

Logical design transforms the conceptual model into structures specific to the chosen database model (typically relational), defining tables, columns, keys, and relationships. This stage applies normalization principles to refine the structure, independent of any specific database system [11].

Physical design addresses how the logical design will be implemented in a specific database management system, considering factors like storage structures, indexing strategies, and access methods. This stage optimizes the design for performance based on anticipated usage patterns [26].

Moving through these levels allows database designers to progressively refine the database structure, addressing different concerns at each stage. As Elmasri and Navathe observe, "The separation of conceptual, logical, and physical design allows database designers to focus on the appropriate level of abstraction at each stage" [11].

For automotive parameter management, this layered approach helps manage the complexity of the domain, ensuring that the resulting database effectively supports both the business requirements (storing and managing parameter configurations) and the technical requirements (performance, scalability, and maintainability).

3 State of the Art

This chapter examines the current state of the art in database version control systems and automotive parameter management. It begins by analyzing existing approaches to software configuration management in the automotive industry, followed by an evaluation of database versioning techniques and their applicability to parameter management systems. The chapter also explores role-based access control models and integration strategies for enterprise systems, establishing the theoretical foundation for the VMAP system design.

3.1 Parameter Management in Automotive Software Development

The complexity of automotive software has grown exponentially in recent decades, with modern vehicles containing up to 100 million lines of code distributed across dozens of electronic control units (ECUs) [27]. This growth has significantly increased the importance and complexity of parameter management in automotive development.

3.1.1 Evolution of Automotive Parameter Management

Parameter management in automotive systems has evolved from simple calibration tables to sophisticated configuration frameworks managing thousands of parameters across multiple vehicle variants. Broy [6] describes the fundamental challenges in automotive software engineering, highlighting that software complexity is driven by the need to address multiple variants, market requirements, and technical functions. The parameter configuration problem is specifically identified as one of the key challenges in this domain.

Early approaches to parameter management relied on specialized tools provided by ECU suppliers, which typically stored parameters in proprietary formats with limited version control capabilities. Pretschner et al. [27] note that these tools evolved from simple memory editors to more sophisticated calibration environments, but remained focused on individual ECUs rather than system-wide parameter management.

Staron [35] describes how AUTOSAR (Automotive Open System Architecture) has contributed to more structured parameter management by defining standard interfaces and component models that separate parameters from implementation. However, the practical implementation of these standards varies across organizations and ECU suppliers, creating integration challenges for comprehensive parameter management.

3.1.2 Challenges in Automotive Parameter Management

The management of parameters in automotive software development presents specific challenges that distinguish it from general software configuration management. Pretschner et al. [27] identify several key challenges related to variability management in automotive software, including the need to maintain multiple parameter configurations for different vehicle variants, markets, and operating conditions.

Broy [6] emphasizes the challenge of managing interdependencies between parameters, noting that changes to one parameter often require coordinated changes to related parameters to maintain system consistency. This creates a need for sophisticated dependency tracking mechanisms that go beyond traditional version control systems.

Another significant challenge relates to validation requirements for parameter changes. Unlike source code, which can be validated through compilation and static analysis, parameters require functional testing to verify their correctness. Pretschner et al. [27] describe how this validation often involves specialized hardware-in-the-loop or vehicle-level testing, creating a significant gap between parameter modification and validation.

Kiencke and Nielsen [19] discuss the specific challenges related to powertrain control parameters, noting the complex interactions between engine control parameters and their effects on vehicle performance, emissions, and fuel economy. These interactions create a need for sophisticated parameter testing and validation processes beyond simple version control.

3.1.3 Current Approaches and Tools

Current parameter management solutions in the automotive industry span a spectrum from general-purpose tools to specialized automotive calibration systems.

Staron [35] discusses how AUTOSAR tools provide standardized interfaces for parameter management in modern automotive systems, but notes that these tools focus primarily on the technical aspects of parameter definition rather than the organizational processes of parameter development and validation through multiple development phases.

Broy [6] identifies the challenges of integrating parameter management into broader software development processes, noting that many organizations maintain separate workflows for software development and parameter calibration. This separation creates coordination challenges, particularly when parameter changes affect multiple software components or require software modifications.

Pretschner et al. [27] discuss how model-based development approaches are increasingly used in automotive development, with parameters linked to model elements to provide traceability and support automated validation. However, they note that the integration between parameter management tools and modeling environments remains incomplete in many organizations.

For database-oriented approaches to parameter management, Bhattacherjee et al. [3] provide a theoretical foundation by examining the principles of dataset versioning. They describe the fundamental trade-offs between storage efficiency and reconstruction performance, which are particularly relevant for systems that must maintain multiple parameter configurations across different development phases.

3.2 Database Version Control Systems

Version control for database content presents distinct challenges compared to traditional source code version control. While source code version control focuses on tracking changes to text files, database version control must address structured data with complex relationships and constraints [3]. This section examines current approaches to database version control and their applicability to automotive parameter management.

3.2.1 Traditional Database Versioning Approaches

Traditional approaches to database versioning fall into several categories, each addressing different aspects of the versioning challenge. Schema evolution tools focus on tracking and managing changes to database structure through migration scripts or schema manipulation languages. Curino et al. [9] describe an approach for automating database schema evolution in information system upgrades, focusing on maintaining data integrity during schema transitions.

Bhattacherjee et al. [3] provide a comprehensive analysis of dataset versioning approaches, identifying a fundamental trade-off between storage and recreation costs. They categorize versioning strategies into several approaches:

1. Version-first approaches maintain complete snapshots of datasets at specific version points, providing simple retrieval of historical states but requiring substantial storage space.

2. Delta-based approaches store only the changes between versions, reducing storage requirements but increasing the computational cost of reconstructing historical states.

3. Hybrid approaches combine elements of both strategies, typically storing periodic full snapshots with incremental deltas between snapshots.

The authors note that the optimal strategy depends on specific usage patterns, particularly the ratio between storage costs and the frequency and complexity of historical data access operations.

Mueller and Müller [25] describe a practical implementation of database versioning between research institutes, highlighting the challenges of maintaining consistency across systems with different update cycles. Their approach uses a combination of schema versioning and data synchronization mechanisms to maintain consistency while supporting independent evolution.

3.2.2 Temporal Database Approaches

Temporal database approaches provide a theoretical foundation for managing time-varying data in database systems. Kulkarni and Michels [22] describe the temporal features introduced in SQL:2011, which formalized support for period data types and temporal tables in the SQL standard. These features enable tracking of both valid time (business time) and transaction time (system time) dimensions, supporting bi-temporal data management.

The valid time dimension represents when facts are true in the modeled reality, independent of when they are recorded in the database. This dimension supports business-oriented temporal queries such as "What was the value of this parameter in a specific phase?" or "When did this parameter change from value A to value B?" [5]. The transaction time dimension represents when facts are recorded in the database, supporting auditability through questions like "Who changed this parameter, and when did they change it?" [22].

Bi-temporal databases combine both dimensions, providing a comprehensive framework for tracking both when changes occurred in the system and when they became effective in the real world [5]. This approach is particularly valuable for regulated industries like automotive development, where both historical accuracy and change auditability are essential for compliance and quality assurance.

Snodgrass [33] provides a comprehensive guide to developing time-oriented database applications in SQL, describing practical techniques for implementing temporal functionality in relational database systems. The author presents various approaches to tracking historical data, including transaction-time tables, valid-time tables, and bi-temporal tables, with practical implementation guidance for each approach.

Biriukov [4] examines practical implementation aspects of bi-temporal databases, highlighting the challenges of schema design, query formulation, and performance optimization. The author notes that domain-specific temporal approaches often provide more practical solutions than generic bi-temporal frameworks, particularly for applications with specialized temporal requirements.

3.2.3 Version Control for Parameter Management

Version control for automotive parameter management presents specific requirements that differ from general database versioning needs. Drawing from the literature, several key requirements can be identified:

Broy [6] discusses the need for version control approaches that align with automotive development processes, which typically follow a structured progression through predefined development phases. Unlike source code versioning, which often follows continuous development with arbitrary version points, parameter versioning must support specific phase-based workflows.

Bhattacherjee et al. [3] examine the trade-offs between different versioning strategies, which are particularly relevant for parameter management systems that must maintain multiple configurations across different development phases. The authors' analysis of storage versus reconstruction costs provides a theoretical foundation for designing efficient parameter versioning systems.

Snodgrass [33] describes techniques for tracking valid-time information in database systems, which aligns with the need to maintain parameter configurations that are valid for specific development phases or vehicle configurations. However, the author's focus on general temporal database approaches does not address the specific requirements of phase-based development.

Bhattacherjee et al. [3] note that domain-specific versioning systems often provide more effective solutions than generic versioning frameworks, particularly for domains with structured development processes and complex entity relationships. This observation supports the development of specialized versioning approaches tailored to automotive parameter management rather than adopting generic temporal database techniques.

3.3 Role-Based Access Control in Enterprise Systems

Role-Based Access Control (RBAC) has become a dominant paradigm for managing access rights in enterprise systems, providing a structured approach to security management that aligns with organizational responsibilities [30]. For automotive parameter management, where different user roles have distinct responsibilities and access requirements, RBAC provides a foundation for implementing appropriate security controls.

3.3.1 RBAC Model and Extensions

The core RBAC model, as defined by Sandhu et al. [30], consists of users, roles, permissions, and sessions. Users are assigned to roles that correspond to job functions, and roles are granted permissions that authorize specific operations on protected resources. This indirect association between users and permissions through roles simplifies security administration while maintaining the principle of least privilege.

Several extensions to the basic RBAC model have been developed to address more complex security requirements. Sandhu et al. [30] describe hierarchical RBAC, which introduces role hierarchies that enable permission inheritance between roles, supporting organizational structures with senior roles inheriting permissions from junior roles.

Sandhu and Bhamidipati [29] present administrative RBAC (ARBAC), which addresses the management of the RBAC system itself, defining who can assign users to roles and modify role permissions. This extension is particularly relevant for enterprise systems where role and permission management is distributed across different administrative domains.

Ferraiolo et al. [12] describe policy-enhanced RBAC, which combines role-based permissions with attribute-based policies to provide context-sensitive access control. This hybrid approach is particularly valuable for systems where access decisions depend on both user roles and context-specific factors such as time, location, or resource attributes.

3.3.2 RBAC in Database Systems

Modern database management systems provide varying levels of support for RBAC principles. Elmasri and Navathe [11] describe the evolution of database security mechanisms from simple user-based privileges to more sophisticated role-based models. Most enterprise database systems now include native support for roles, user-role assignments, and permission management through SQL statements like GRANT and REVOKE.

Obe and Hsu [26] detail PostgreSQL's implementation of RBAC concepts, including role hierarchies through role inheritance, permission management through fine-grained privileges, and row-level security policies for content-based access control. These capabilities provide a foundation for implementing domain-specific access control models on top of the database system's native security features.

However, database-level RBAC implementations typically focus on controlling access to database objects like tables, views, and functions, rather than providing application-level access control that considers domain-specific entities and operations. For complex applications like automotive parameter management, database-level RBAC must be complemented with application-level access control logic that maps domain-specific concepts to database operations [12].

3.3.3 Access Control for Parameter Management

Access control for automotive parameter management presents specific requirements that extend beyond basic RBAC models. Drawing from the literature, several key access control requirements can be identified:

Sandhu et al. [30] provide the theoretical foundation for role-based access control, which aligns with the organizational structure of automotive development teams. Different roles such as parameter engineers, module developers, and system integrators require different access rights to parameter data.

Ferraiolo et al. [12] describe policy-enhanced RBAC, which combines role-based permissions with attribute-based policies. This hybrid approach is particularly relevant for parameter management, where access rights may depend on both user roles and attributes of the parameters being accessed, such as their development phase or module assignment.

Hu et al. [16] discuss practical aspects of implementing and managing policy rules in attribute-based access control, providing insights into the challenges of combining role-based and attribute-based approaches. Their work highlights the importance of balancing security requirements with usability considerations, which is particularly relevant for parameter management systems used by diverse stakeholder groups.

Sandhu and Bhamidipati [29] address the administrative aspects of RBAC, which are important for parameter management systems where access control administration may be distributed across different organizational units. Their ARBAC97 model provides a framework for delegating administrative responsibilities while maintaining central governance.

3.4 Database Integration with Enterprise Systems

Integration between database systems and enterprise applications presents significant challenges in automotive development environments, where parameter management must interact with numerous other systems across the development lifecycle. Effective integration strategies must address both technical interoperability and semantic consistency while maintaining performance and security [15].

3.4.1 Enterprise Integration Patterns

Enterprise integration patterns, as documented by Hohpe and Woolf [15], provide a catalog of solutions for common integration challenges. These patterns address various aspects of system integration, including messaging styles, messaging channels, message construction, and message transformation.

For database-centric applications like parameter management systems, several integration patterns are particularly relevant. Fowler [13] describes the Repository pattern, which provides a structured approach to data access, abstracting the database implementation details behind a domain-focused interface. This abstraction simplifies integration by providing a stable API for other systems to interact with the parameter repository.

Fowler [13] also documents the Data Transfer Object (DTO) pattern, which addresses the challenge of transferring data between systems with different data models. By defining specialized objects for inter-system communication, this pattern enables consistent data exchange while isolating each system's internal representation.

The Canonical Data Model pattern, as described by Hohpe and Woolf [15], establishes a common data representation across multiple systems, simplifying data transformation and ensuring consistent interpretation. This pattern is particularly valuable for parameter management, where the same parameter concepts may be represented differently in various systems across the development lifecycle.

3.4.2 Database Synchronization Approaches

Database synchronization presents specific challenges when integrating parameter management systems with other enterprise data sources. Mueller and Müller [25] describe approaches to database versioning and synchronization between research institutes, highlighting the challenges of maintaining consistency across systems with different update cycles.

Bhattacherjee et al. [3] discuss the principles of dataset versioning, which are relevant for synchronization between parameter management systems and other enterprise databases. Their analysis of the trade-offs between storage and recreation costs provides insights into designing efficient synchronization mechanisms that minimize both data transfer volumes and processing overhead.

Seenivasan and Vaithianathan [32] examine change data capture (CDC) techniques, which enable incremental synchronization by identifying and propagating only changed data between systems. These techniques reduce synchronization overhead compared to full dataset transfers but require reliable change detection mechanisms and careful handling of interdependent changes.

Kleppmann and Beresford [20] address the challenges of conflict resolution in distributed data systems, which are relevant for parameter management systems that must synchronize with multiple enterprise data sources. Their work on conflict-free replicated data types provides theoretical foundations for designing synchronization mechanisms that maintain consistency across distributed systems.

3.5 Summary and Research Gaps

The review of existing literature reveals several research gaps in the domain of database systems for automotive parameter management:

Current database versioning approaches, as described by Bhattacherjee et al. [3] and Snodgrass [33], provide general frameworks for managing time-varying data but do not specifically address the phase-based development processes common in automotive parameter management. There is a need for specialized versioning approaches that align directly with automotive development workflows while providing the traceability and auditability required for regulatory compliance.

The RBAC models described by Sandhu et al. [30] and Ferraiolo et al. [12] provide a foundation for access control but require extensions to address the specific requirements of parameter management, where access rights depend on both organizational roles and parameter-specific attributes such as module assignment and development phase.

Integration approaches documented by Hohpe and Woolf [15] and Mueller and Müller [25] provide general patterns for system integration but do not specifically address the challenges of integrating parameter management systems with automotive-specific enterprise systems such as parameter definition databases and vehicle configuration databases.

These research gaps highlight the need for domain-specific solutions that combine insights from database version control, access control models, and enterprise integration patterns with specialized knowledge of automotive development processes. The VMAP system addresses these gaps by developing a database architecture tailored to the specific requirements of automotive parameter management, as will be detailed in subsequent chapters.

4 Methodology and Concept Development

This chapter presents the systematic approach taken in designing the Variant Management and Parametrization (VMAP) system. The methodology follows established software engineering principles to address the complex requirements of automotive parameter management. Beginning with a requirements analysis, the chapter proceeds to detail the conceptual architecture design, data model, validation mechanisms, and integration approaches developed to ensure system robustness and compatibility with existing enterprise infrastructure.

4.1 Requirements Analysis

The foundation of the VMAP system design was a comprehensive requirements analysis conducted through a series of structured interviews with stakeholders, detailed examination of the existing Excel-based process, and workshops with domain experts. This multi-faceted approach, following Sommerville's framework for requirements engineering, ensured that both functional and non-functional requirements would be thoroughly identified and prioritized [34].

4.1.1 Functional Requirements

The primary functional requirements were derived from direct observation of engineers' current Excel-based workflow combined with semi-structured interviews conducted with module developers and documentation specialists. Through this process, several critical requirements emerged for the VMAP system.

The system must support the hierarchical organization of parameters within Electronic Control Units (ECUs), Modules, and Parameter IDs (PIDs), mirroring the domain-specific structure of automotive electronic systems as described by Staron [35]. This hierarchical organization is essential for maintaining the logical structure of vehicle parameters and aligning with established engineering practices.

Users must be able to create variants for parameters with specific code rules determining their applicability, and define segments representing modified parameter values. If no segment exists, the system must default to Parameter Definition Database values—an approach that allows efficient storage by tracking only modifications rather than duplicating unchanged parameters, aligning with Bhattacherjee's principles of dataset versioning [3].

The system must track parameter values across four distinct release phases: Phase1, Phase2, Phase3, and Phase4, with changes in earlier phases propagating to later phases unless explicitly overridden. This phase-based approach represents a domain-specific adaptation particularly suited to automotive software development cycles as identified in Broy's research on automotive software engineering challenges [6].

All modifications require comprehensive logging with user information, timestamp, and detailed change data, supporting regulatory compliance and enabling parameter evolution tracking. Through the stakeholder interviews, it was determined that the system must also provide functionality to create parameter configuration snapshots at specific points, particularly at phase transitions, for documentation purposes—a capability identified as essential for quality assurance and regulatory compliance in automotive software development by Staron [35].

4.1.2 Integration with External Systems

The stakeholder interviews and process analysis revealed that VMAP must integrate with two critical external enterprise systems: the Parameter Definition Database (PDD) and the Vehicle Configuration Database (VCD).

The Parameter Definition Database (PDD) serves as the authoritative source for the hierarchical structure of automotive electronic systems, containing definitions of ECUs, Modules, PIDs, and baseline parameter configurations. As noted by Pretschner et al. [27], maintaining this hierarchical structure is essential for automotive software development. The workshops with domain experts confirmed that while VMAP will manage parameter variants and customizations, it must rely on PDD for the underlying parameter definitions and structural relationships, requiring a robust synchronization mechanism to maintain consistency between the systems.

The Vehicle Configuration Database (VCD) contains comprehensive vehicle specifications and configuration codes that determine which parameter variants apply to specific vehicle configurations. Integration with this system is necessary for two critical functions: validating the boolean code rules associated with parameter variants to ensure they reference valid vehicle codes, and supporting parameter file generation for specific vehicle configurations by resolving the applicable parameter variants based on vehicle codes. This integration requirement aligns with Staron's analysis of automotive software architectures, which emphasizes the importance of configuration management in supporting variant-rich vehicle platforms [35].

These integration requirements necessitated careful consideration of data synchronization approaches, leading to an exploration of different strategies for maintaining consistency between VMAP and these external systems while minimizing performance impact and complexity.

4.1.3 User Role Requirements

A systematic analysis of the current Excel-based workflow, coupled with contextual inquiries with engineering teams, identified four distinct user roles with specific access requirements. This analysis included shadowing users in their daily work, documenting their tasks and access patterns, and conducting structured interviews to validate the observed patterns.

Module developers require write access to parameters within their assigned modules, with the ability to create and modify variants and segments. Documentation specialists need access to frozen data for documentation, comparison capabilities between phases, and comprehensive change history access. System administrators require comprehensive control over user management, release phases, and special operations like variant deletion and phase freezing. Read-only users need view access to all parameter data with parameter file generation capabilities but no modification rights.

These roles were defined based on the principle of least privilege as described by Sandhu [30], ensuring users have access only to functionality required for their specific responsibilities. This enhances system security while simplifying the user experience by presenting only relevant options.

4.1.4 Data Management Requirements

The system must maintain distinct parameter versions across different release phases, allowing simultaneous work on multiple phases while enabling access to parameter values from any point in the development lifecycle. As highlighted by Elmasri and Navathe [11], data integrity requires maintaining referential integrity across all related entities, particularly ensuring variants and segments associate with valid parameters.

Multi-dimensional parameter support is essential for complex automotive parameters such as mapping tables. Operations modifying multiple related entities must function as atomic transactions to maintain data consistency—particularly important for phase transitions where numerous parameters, variants, and segments may change simultaneously, a requirement that aligns with Bhattacherjee's research on dataset versioning approaches [3].

Query performance analysis, based on projected usage patterns from the current Excel-based process, identified critical query paths including parameter retrieval by ECU, module, PID, release phase, and parameter name. These requirements influenced schema design decisions regarding normalization and indexing strategies to optimize common query patterns.

4.2 Use Case Modeling

Following the requirements gathering process, use case modeling was employed to formalize the system's functional requirements from a user perspective. This approach, as described by Jacobson [17], provides a structured way to represent the system's capabilities and the interactions between users and the system.

The use case diagram in Figure 4.1 illustrates the primary actors and their interactions with the VMAP system. Four primary actor types are identified, corresponding to the user roles established during requirements analysis: Module Developers, who create and modify parameter variants; Documentation Team members, who access parameter data for documentation purposes; Administrators, who manage system settings and user access; and Read-Only Users, who view parameter data without making modifications.

The diagram demonstrates how these actors interact with key system functionalities. Module Developers primarily interact with variant creation and modification use cases, while having limited access to parameter viewing and comparison features. Documentation Team members focus on viewing frozen configurations, comparing parameters across phases, and accessing change history. Administrators have access to all system functions, including user management, release configuration, and system settings. Read-Only Users are limited to viewing parameters and generating parameter files.

This use case model provides a clear visual representation of the system's scope and functionality, serving as a bridge between user requirements and technical implementation. By mapping user roles to specific system functions, the model ensures that the database design will support all required user interactions while maintaining appropriate access controls.

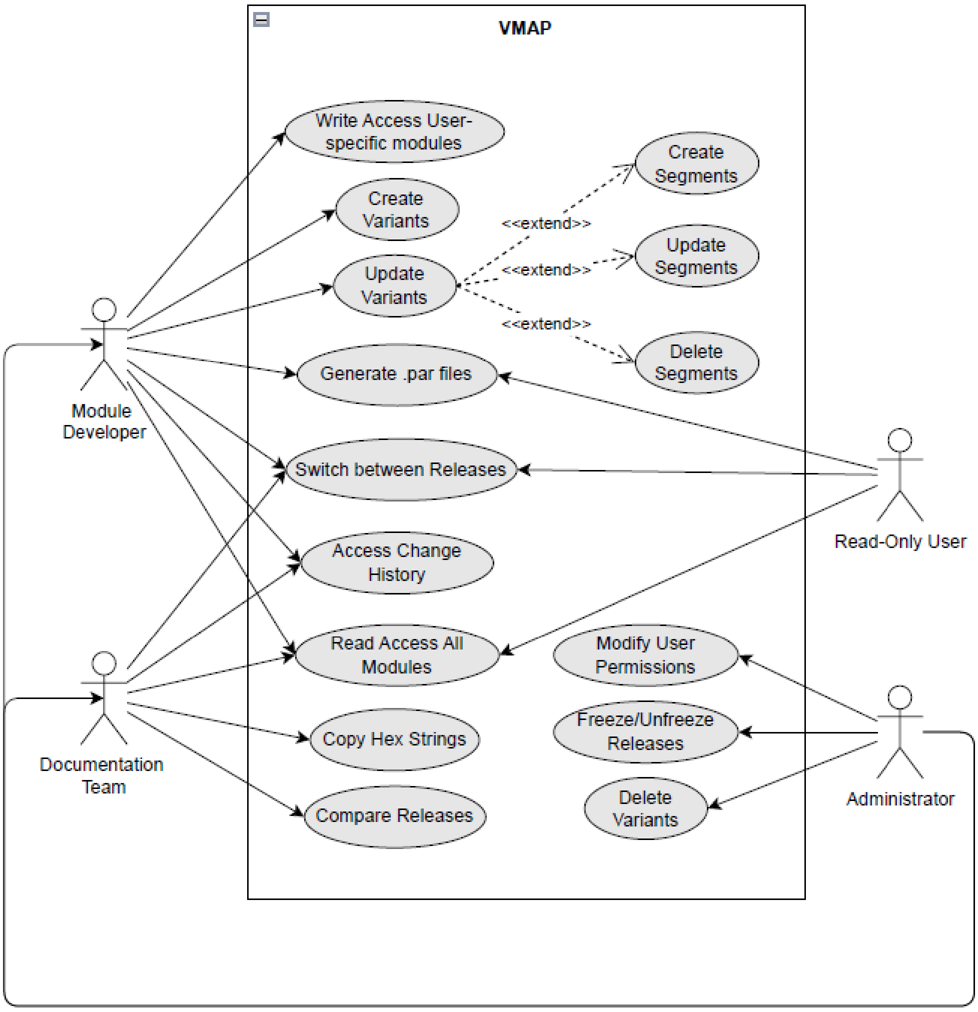


Figure 4.1: VMAP System Use Case Diagram

4.3 User Management Approaches

Based on the identified user role requirements, two distinct approaches to user management were considered for the VMAP system: a traditional role-based approach and a hybrid role-permission approach. Each approach offers different advantages in terms of flexibility, administrative complexity, and alignment with organizational needs.

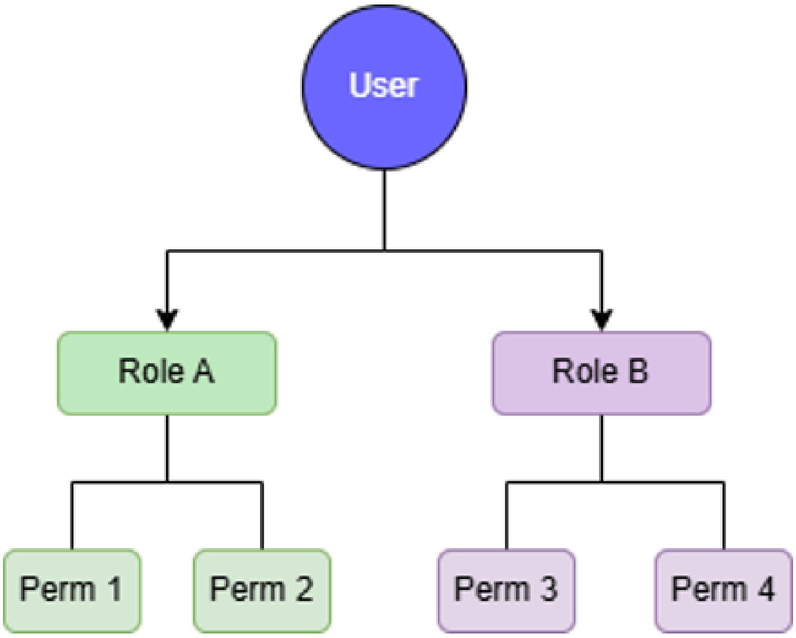


Figure 4.2: Traditional Role-Based Access Control Approach

4.3.1 Traditional Role-Based Approach

The traditional role-based approach, illustrated in Figure 4.2, assigns users to predefined roles that contain fixed sets of permissions, following the classic Role-Based Access Control (RBAC) model described by Sandhu [30]. In this approach, each user is assigned one or more roles (Administrator, Module Developer, Documentation Team, Read-Only User), and all permissions are granted through these role assignments without individual permission adjustments.

This approach offers administrative simplicity, as user management involves only assigning appropriate roles rather than configuring individual permissions. The role structure also provides clear organizational alignment, with roles directly corresponding to job functions within the development process. From an implementation perspective, this approach simplifies permission checking, typically requiring only verification of role membership rather than individual permission verification.

A key limitation of this approach is its reduced flexibility for accommodating exceptions or specialized access requirements. If a user requires a subset of permissions that doesn't align with existing roles, administrators must either create a new role specifically for that user or grant a role with more permissions than strictly necessary, potentially compromising the principle of least privilege [30].

4.3.2 Hybrid Role-Permission Approach

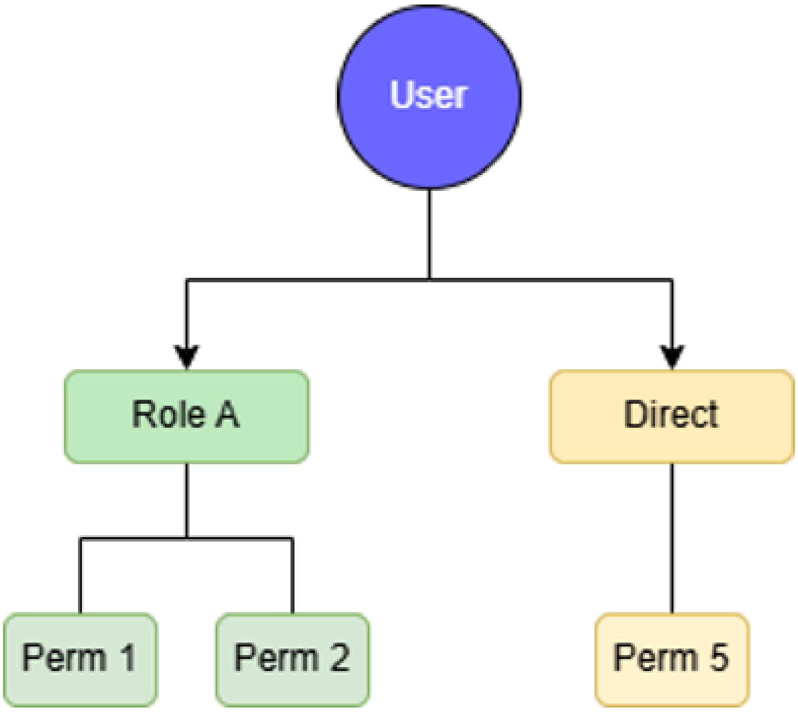


Figure 4.3: Hybrid Role-Permission Access Control Approach

The hybrid approach, illustrated in Figure 4.3, combines role-based permissions with direct permission assignments, similar to the model described by Ferraiolo et al. [12]. In this approach, users are assigned to primary roles defining their core permissions, but additional permissions can be granted on a per-user basis to address exceptional cases or specialized responsibilities.

This approach offers greater flexibility for accommodating exceptions without creating specialized roles, essential in environments where organizational structures evolve over time. It provides more granular permission control, allowing precise tailoring of access rights to individual responsibilities. However, this flexibility comes at the cost of increased administrative complexity, as both roles and individual permissions must be managed.

The hybrid approach is particularly valuable in the automotive parameter management context, where development responsibilities can vary between projects and temporary access adjustments may be needed for specific tasks or during transition periods. The approach balances structured role assignments with the flexibility to accommodate evolving access requirements, a common scenario in complex engineering environments like automotive development.

4.4 Parameter Synchronization Approaches

Integration with the Parameter Definition Database (PDD) represents a critical aspect of the VMAP system, requiring careful consideration of synchronization approaches. Two different conceptual approaches were explored for maintaining parameter data across the release phases: the change-based approach and the phase-based approach.

4.4.1 Change-Based Synchronization Approach

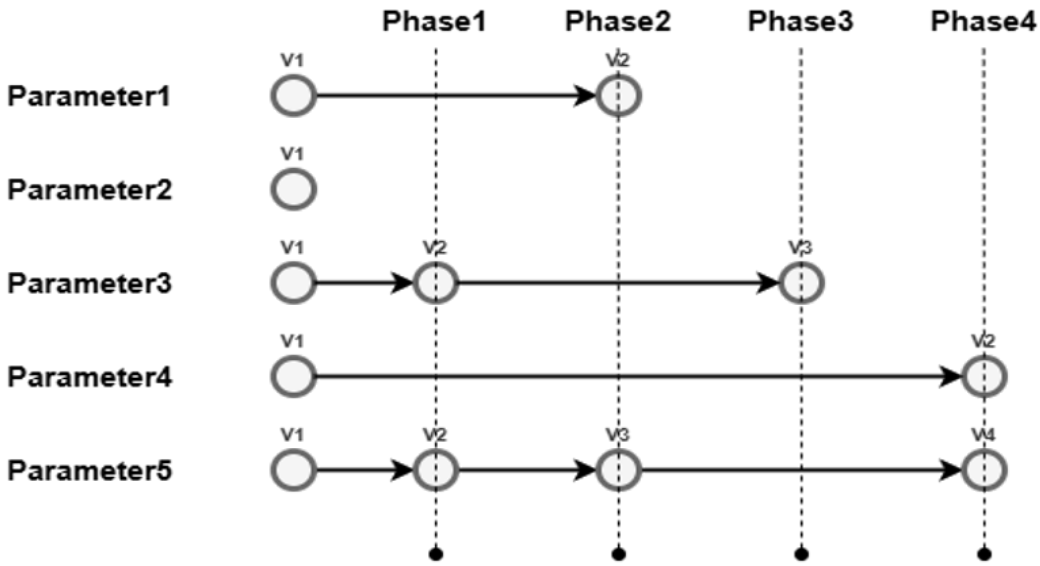


Figure 4.4: Change-Based Parameter Synchronization Approach

The change-based approach, illustrated in Figure 4.4, maintains parameter values by recording changes between phases rather than storing complete parameter sets for each phase. In this model, parameters are initially created before the first phase (Phase1), and subsequent modifications are recorded as change entries associated with specific phases.

When a parameter changes in a later phase (like Phase2), only the specific change is recorded rather than creating a new complete copy of the parameter. As shown in the figure, Parameter1 is created before Phase1 with version V1, then modified in Phase2 (creating version V2), but remains unchanged in Phase3 and Phase4. Similarly, Parameter3 changes in Phase2 and Phase3, while Parameter5 changes in every phase except Phase3.

This approach is conceptually aligned with traditional version control systems as described by Bhattacherjee et al. [3], where efficiency is achieved by storing only the differences between versions rather than complete copies. The approach potentially offers storage efficiency advantages by minimizing data duplication across phases, which could be significant for parameter sets containing thousands of entries.

However, this approach introduces conceptual complexity for retrieving parameter values in a specific phase. To determine a parameter's value for a given phase, the system must identify the most recent version of that parameter up to and including the target phase. This reconstruction process introduces additional processing steps compared to direct parameter retrieval.

4.4.2 Phase-Based Synchronization Approach

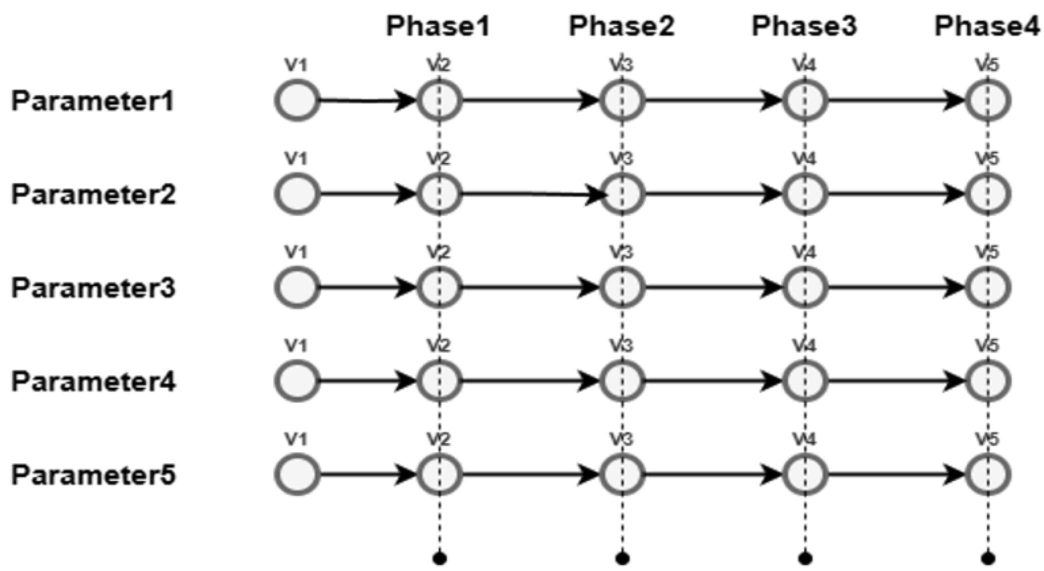


Figure 4.5: Phase-Based Parameter Synchronization Approach

The phase-based approach, illustrated in Figure 4.5, maintains complete parameter sets for each phase independently. When parameters are initially created before Phase1, each parameter has its specific version (V1). When transitioning to a new phase, all parameters are copied forward, even if they haven't changed. If a parameter is subsequently modified in the new phase, it receives a new version specific to that phase.

As shown in the figure, each parameter exists in every phase with a phase-specific version, regardless of whether the parameter value actually changed between phases.

This creates a clear separation between phases, with each phase maintaining its complete parameter configuration independently.

This approach aligns more directly with the phase-oriented structure of automotive development described by Broy [6], where distinct development milestones form the primary organizational principle. The approach simplifies conceptual understanding and parameter retrieval, as parameters for a specific phase can be accessed directly without reconstructing their values from change history.

The phase-based approach also simplifies phase inheritance by copying parameter configurations forward during phase transitions, allowing subsequent modifications in each phase without affecting previous phases. This copying mechanism preserves the integrity of phase data while supporting the automotive development process, where configurations stabilize progressively through successive phases.

The primary consideration with this approach is the increased storage requirements, as parameters are duplicated across phases even when they haven't changed. However, this trade-off potentially provides benefits in terms of conceptual clarity, query simplicity, and alignment with the automotive development workflow.

4.5 Database System Considerations

Selecting an appropriate database management system for VMAP required consideration of different options against the specific requirements of automotive parameter management. Four major relational database systems were considered as potential platforms for the VMAP implementation: PostgreSQL, Oracle, Microsoft SQL Server, and MySQL.

4.5.1 Database System Requirements

The requirements analysis identified several critical database capabilities needed for effective parameter management:

1. Support for complex data types, including arrays for multi-dimensional parameters and structured types for variant definitions.

2. Robust transaction support for maintaining data consistency during operations affecting multiple related entities.

3. Advanced indexing capabilities to optimize the performance of common query patterns, particularly parameter retrieval across different dimensions.

4. Extensibility for implementing domain-specific operations and validation rules.

5. Comprehensive access control mechanisms supporting the role-based security model.

6. Efficient storage and retrieval of historical data for audit and traceability purposes.

These requirements guided the evaluation of different database systems, focusing on their respective strengths and limitations in addressing the specific needs of automotive parameter management.

4.5.2 Comparative Analysis of Database Systems

Table 4.1 presents a comparative analysis of the four database systems considered for the VMAP implementation, evaluating each against criteria relevant to automotive parameter management.

The comparative analysis revealed different strengths among the database systems. PostgreSQL offers excellent support for complex data types and extensibility, particularly valuable for representing multi-dimensional parameters and implementing domain-specific operations. Oracle provides robust enterprise features with sophisticated optimization capabilities but introduces licensing complexity. SQL Server offers strong integration with Microsoft technologies, while MySQL provides simplicity but has limitations for complex data management requirements.

This analysis provides a foundation for database system selection, considering both technical capabilities and practical factors such as licensing and total cost of ownership. The implementation chapter will detail the specific database system selected and how its capabilities are leveraged in the VMAP implementation.

4.6 Entity-Relationship Model

Based on the requirements analysis and architectural considerations, a comprehensive entity-relationship (ER) model was developed to capture the complex relationships between system entities. This model follows the approach described by Chen [7], providing a conceptual foundation for the database implementation.

Table 4.1: Comparison of Database Systems for Automotive Parameter Management

Feature	PostgreSQL	Oracle	SQL Server	MySQL
Complex Data Types	Excellent support for arrays, JSON, custom types [26]	Good support, additional licensing for advanced features [1]	Limited built-in support, extensions required [2]	Limited support, improved in recent versions [39]
Transaction Support	Comprehensive with serializable isolation [26]	Excellent with advanced options [1]	Robust support with multiple isolation levels [2]	Limited in some storage engines [31]
Indexing Capabilities	Diverse index types including GIN for text search [26]	Advanced indexing with optimizer hints [1]	Solid capabilities with columnstore indexes [2]	Basic indexing with some limitations [31]
Extensibility	Highly extensible with custom types and functions [26]	Extensible with proprietary mechanisms [1]	Extensible through .NET integration [2]	Limited extensibility [39]
Access Control	Fine-grained with role-based mechanisms [26]	Comprehensive with advanced security features [1]	Strong integration with Active Directory [2]	Basic capabilities with plugin architecture [39]
Licensing	Open source, PostgreSQL License [26]	Commercial, complex licensing model [1]	Commercial with edition-based pricing [2]	Dual licensing: GPL and commercial [39]

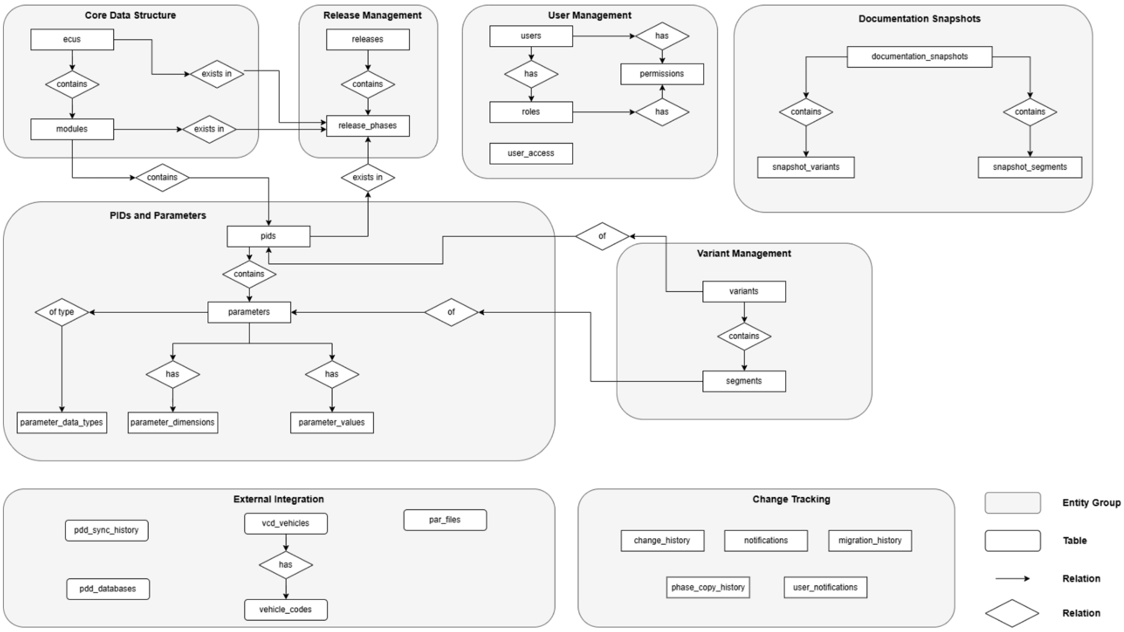


Figure 4.6: Comprehensive Entity-Relationship Diagram for the VMAP System

Figure 4.6 illustrates the complete entity-relationship model for the VMAP system, showing the logical organization of entities into functional groups and the relationships between them. The diagram captures the hierarchical structure of automotive parameter data, the phase-based versioning approach, the role-permission access control model, and the mechanisms for external system integration and change tracking.

4.6.1 Core Data Entities

The ER model includes several categories of entities representing different aspects of the system:

User management entities include Users, Roles, Permissions, and their relationships, implementing a role-permission model for access control. These entities support the authentication and authorization requirements identified during the requirements analysis.

Release management entities encompass Releases, Release Phases, and ECU Phase mappings, providing the foundation for the phase-based version control approach. These entities support the automotive development cycle with its distinct phases and milestones.

Parameter structure entities include ECUs, Modules, PIDs, Parameters, and Parameter Dimensions, representing the hierarchical organization of vehicle electronic systems as described by Staron [35]. These entities capture both the structure and characteristics of parameters, including multi-dimensional parameters like maps and tables.

Variant management entities comprise Variants, Segments, and their relationships to parameters, implementing the core parameter customization functionality. These entities support the creation of parameter variations for different vehicle configurations and operating conditions.

Documentation entities include Documentation Snapshots, Snapshot Variants, and Snapshot Segments, supporting the preservation of historical parameter states for documentation and regulatory compliance. These entities enable the creation of complete parameter configuration snapshots at specific development milestones.

Integration entities consist of Synchronization Records, Vehicle Configurations, and Parameter File Records, supporting connectivity with the Parameter Definition Database and Vehicle Configuration Database. These entities track synchronization operations and store the necessary data for variant resolution and parameter file generation.

Audit entities encompass Change History, Transaction Records, and Phase Copy History, providing comprehensive traceability for all significant operations within the system. These entities support both regulatory compliance and diagnostic capabilities for investigating parameter evolution.

4.6.2 Relationship Structure

The relationships between entities in the ER model reflect the complex interactions between different aspects of automotive parameter management. Key relationships include:

Hierarchical relationships between ECUs, Modules, PIDs, and Parameters, representing the structural organization of automotive electronic systems. These relationships enforce the domain-specific hierarchy while supporting navigation from higher-level entities to their components.

Many-to-many relationships between parameters and phases, implemented through direct association rather than temporal versioning. This structure supports the phase-based versioning approach, allowing efficient retrieval of parameters for specific phases.

Complex relationships between variants, parameters, and segments, capturing the parameter customization process. These relationships ensure that segments are associated with valid variants and parameters while supporting efficient resolution of effective parameter values based on vehicle configuration.

Temporal relationships for audit and history entities, capturing the evolution of parameter configurations over time. These relationships support both regulatory compliance and diagnostic capabilities, allowing reconstruction of parameter states at specific points in time.

4.6.3 Normalization and Optimization

The ER model was developed using data normalization principles to minimize redundancy while maintaining data integrity, following the approach described by Codd [8]. The model generally adheres to Third Normal Form (3NF), ensuring that non-key attributes depend on the primary key rather than on other non-key attributes.

Strategic denormalization was considered in specific areas to optimize performance for common operations, following the principles described by Molinaro [24]. For example, certain frequently accessed attributes from parent entities might be duplicated in child entities to reduce join operations in common queries, providing performance benefits that outweigh the controlled redundancy.

4.7 Validation Mechanisms

To ensure data integrity and consistency, multiple validation mechanism layers were conceptualized for the VMAP system, from basic constraints to sophisticated business rule validation. These mechanisms work together to maintain parameter data quality and reliability throughout the system lifecycle.

4.7.1 Data Integrity Constraints

Database-level constraints were identified as the foundation for enforcing basic integrity rules, following the principles described by Elmasri and Navathe [11]. These constraints include primary key constraints ensuring unique entity identifiers, foreign key constraints maintaining referential integrity between related entities, not-null constraints ensuring required fields contain values, unique constraints preventing duplicate values in specified columns, and check constraints enforcing domain-specific rules such as valid date ranges and parameter value ranges.

These constraints are designed into the database schema as integral parts of entity definitions, ensuring consistent enforcement throughout the system regardless of access path. By implementing constraints at the database level rather than in application code, VMAP ensures that all data modifications adhere to fundamental integrity rules regardless of the source of those modifications.

4.7.2 Business Rule Validation

Domain-specific business rules were conceptualized to be implemented through database triggers and stored procedures, providing a second layer of validation beyond basic constraints. These rules include parameter range validation automatically checking modified values against defined minimum and maximum bounds, phase status validation preventing modifications to frozen phases, segment validation ensuring segments reference valid parameters and variants, and user access validation ensuring users can only modify parameters, variants, and segments for modules to which they have been granted access.

4.7.3 Conflict Resolution Strategies

In a multi-user environment, concurrent modifications can create conflicts. Several strategies were conceptualized to detect and resolve these conflicts, maintaining data consistency while supporting collaborative parameter management.

For web-based interactions, optimistic concurrency control using version timestamps allows multiple users to view the same data concurrently, detecting conflicts only when updates collide. This approach aligns with the principles described by Bhattacherjee et al. [3], optimizing for the common case where conflicts are rare while still ensuring data integrity.

When changes propagate from one phase to the next, conflicts can arise if the target phase has already been modified. Explicit conflict resolution mechanisms compare the source variant or segment with existing target configurations during phase propagation operations. When conflicts are detected, resolution options allow users to make informed decisions: override the target with source values, preserve target values, or merge values based on specified rules.

4.7.4 Audit and Traceability Mechanisms

Comprehensive audit and traceability mechanisms were identified as essential for regulatory compliance and quality assurance in automotive parameter management. The core of this capability is the change history tracking mechanism, which automatically captures both before and after states for entity modifications. For each change, the system records the entity being modified, the type of change, the user making the change, the timestamp, and detailed before/after values.

To optimize performance, selective filtering of change data was conceptualized, excluding non-essential fields such as timestamps and large binary data. Additionally, asynchronous audit recording for bulk operations was considered, to reduce the performance impact on high-volume operations while ensuring that all changes are eventually recorded.

Beyond change tracking, specialized audit mechanisms were conceptualized for specific scenarios: phase transition logging to record all phase propagation operations, freeze operation logging to record phase freeze and unfreeze operations, user access logging to capture authentication and authorization events, and integration operation logging to record all external system interactions.

5 Implementation

This chapter presents the technical implementation of the VMAP system conceptual architecture described in Chapter 4. Rather than exhaustively cataloging implementation details, the discussion focuses on key architectural components, highlighting technical decisions that enable efficient parameter versioning in automotive software development.

5.1 Database Structure Implementation

The database implementation transforms abstract entities and relationships into concrete PostgreSQL structures, implementing the hierarchical organization of automotive electronic systems through four primary entity types: ECUs, Modules, PIDs, and Parameters.

5.1.1 Core Data Entities

The ECU and Module entities form the top levels of the hierarchy, implemented with a many-to-many relationship reflecting the reality that the same logical module may exist across multiple ECUs:

```sql
CREATE TABLE ecus (
    ecu_id INTEGER PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    byte_order VARCHAR(50),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    external_id INTEGER UNIQUE -- PDD reference ID
);

CREATE TABLE modules (
    module_id INTEGER PRIMARY KEY,
    shortcut VARCHAR(50) NOT NULL,
    name VARCHAR(255) NOT NULL,
```

```sql
    kind VARCHAR(255),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    external_id INTEGER UNIQUE -- PDD reference ID
);

-- Link ECUs and Modules - A module can exist in multiple ECUs
CREATE TABLE ecu_modules (
    ecu_id INTEGER REFERENCES ecus(ecu_id) ON DELETE CASCADE,
    module_id INTEGER REFERENCES modules(module_id) ON DELETE CASCADE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (ecu_id, module_id)
);
```

Listing 5.1: ECU and Module Table Implementation

The junction table `ecu_modules` implements this many-to-many relationship following the foreign key pattern described by Elmasri and Navathe [11]. Each entity includes an `external_id` attribute to maintain mapping with the Parameter Definition Database, facilitating integration and synchronization.

PIDs and Parameters constitute the lower hierarchy levels:

```sql
CREATE TABLE pids (
    pid_id BIGINT PRIMARY KEY,
    ecu_id INTEGER REFERENCES ecus(ecu_id) ON DELETE CASCADE,
    module_id INTEGER REFERENCES modules(module_id) ON DELETE CASCADE,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    external_id INTEGER UNIQUE, -- PDD reference ID
    -- Ensure the ECU-Module combination exists
```

```
    CONSTRAINT pid_ecu_module_fk FOREIGN KEY (ecu_id, module_id)
    REFERENCES ecu_modules(ecu_id, module_id)
);

CREATE TABLE parameters (
    parameter_id BIGINT PRIMARY KEY,
    pid_id BIGINT REFERENCES pids(pid_id) ON DELETE CASCADE,
    ecu_id INTEGER,
    phase_id INTEGER,
    name VARCHAR(255) NOT NULL,
    parameter_name VARCHAR(255),
    type_id INTEGER REFERENCES parameter_data_types(data_type_id),
    array_definition VARCHAR(50),
    position INTEGER,
    factor DECIMAL,
    unit VARCHAR(50),
    bias_offset DECIMAL,
    is_active BOOLEAN DEFAULT true,
    external_id INTEGER, -- PDD reference ID
    FOREIGN KEY (ecu_id, phase_id) REFERENCES ecu_phases(ecu_id, phase_id)
);
```

Listing 5.2: PID and Parameter Table Implementation

The PID table includes a compound foreign key constraint ensuring each PID references a valid ECU-Module combination, preventing orphaned PIDs. The parameters table incorporates strategic denormalization with direct references to `ecu_id` and `phase_id` alongside the PID foreign key. While introducing some redundancy, this approach significantly improves performance for parameter queries by release phase, a common operation in the system. As noted by Bhattacherjee et al. [3], strategic denormalization can substantially improve query performance when data access patterns favor specific traversal paths.

A particularly challenging aspect was implementing multi-dimensional parameters, common in automotive applications for lookup tables and characteristic curves:

```
CREATE TABLE parameter_dimensions (
    dimension_id BIGINT PRIMARY KEY,
    parameter_id BIGINT REFERENCES parameters(
      ↳parameter_id) ON DELETE CASCADE,
    dimension_index INTEGER NOT NULL,
    default_value NUMERIC NOT NULL,
    external_id INTEGER, -- PDD reference ID
    UNIQUE (parameter_id, dimension_index)
);
```

Listing 5.3: Parameter Dimension Table Implementation

This implementation follows a modified entity-attribute-value (EAV) pattern addressing the need for flexible dimensionality while maintaining query performance. As noted by Nadkarni et al. [**?**], traditional EAV models can suffer from performance limitations, but careful design constraints can mitigate these issues. The `dimension_index` field provides an ordered structure to dimensions, enabling efficient representation of arrays and matrices.

5.1.2 User Management and Access Control

The user management implementation realizes the role-based access control (RBAC) model with extensions supporting module-specific permissions:

```
CREATE TABLE users (
    user_id BIGINT PRIMARY KEY,
    emp_id VARCHAR(50) UNIQUE NOT NULL,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    status VARCHAR(50),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP
);

CREATE TABLE roles (
    role_id INTEGER PRIMARY KEY,
```

```sql
    name VARCHAR(255) UNIQUE NOT NULL,
    description TEXT
);

CREATE TABLE permissions (
    permission_id INTEGER PRIMARY KEY,
    name VARCHAR(255) UNIQUE NOT NULL,
    description TEXT
);

CREATE TABLE role_permissions (
    role_id INTEGER REFERENCES roles(role_id) ON DELETE CASCADE,
    permission_id INTEGER REFERENCES permissions(permission_id) ON DELETE CASCADE,
    granted_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    granted_by BIGINT REFERENCES users(user_id),
    PRIMARY KEY (role_id, permission_id)
);

CREATE TABLE user_roles (
    user_id BIGINT REFERENCES users(user_id) ON DELETE CASCADE,
    role_id INTEGER REFERENCES roles(role_id) ON DELETE CASCADE,
    granted_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    granted_by BIGINT REFERENCES users(user_id),
    PRIMARY KEY (user_id, role_id)
);
```

Listing 5.4: Core RBAC Table Implementation

This implementation follows the RBAC0 model defined by Sandhu et al. [30], providing users, roles, and permissions with their many-to-many relationships. The inclusion of `granted_at` and `granted_by` fields in junction tables extends the basic model with audit information, addressing accountability requirements common in regulated industries like automotive development [35].

The RBAC model is extended with module-based access control:

```
CREATE TABLE user_access (
    user_id BIGINT REFERENCES users(user_id) ON DELETE CASCADE,
    ecu_id INTEGER REFERENCES ecus(ecu_id) ON DELETE CASCADE,
    module_id INTEGER REFERENCES modules(module_id) ON DELETE CASCADE,
    write_access BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    created_by BIGINT REFERENCES users(user_id),
    updated_by BIGINT REFERENCES users(user_id),
    PRIMARY KEY (user_id, ecu_id, module_id),
    CONSTRAINT user_access_ecu_module_fk FOREIGN KEY (ecu_id, module_id)
    REFERENCES ecu_modules(ecu_id, module_id)
);
```

Listing 5.5: Module-Based Access Control Implementation

This implementation combines attributes of both role-based and attribute-based access control, creating what Ferraiolo et al. describe as policy-enhanced RBAC [12]. The `user_access` table establishes a three-way relationship between users, ECUs, and modules, with a boolean flag distinguishing between read and write access. The constraint ensures access is granted only for valid ECU-module combinations, enforcing structural integrity. This approach implements the principle of least privilege, allowing administrators to grant write access specifically to modules developers are responsible for while maintaining read access across all modules.

5.1.3 Release Management Implementation

The release management implementation realizes the phase-based versioning approach, providing a flexible framework for managing parameter evolution across development phases:

```
CREATE TABLE releases (
    release_id INTEGER PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE, -- e.g.,
      ↳"24.1", "24.3"
    description TEXT,
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    created_by BIGINT REFERENCES users(user_id)
      ↳,
    updated_by BIGINT REFERENCES users(user_id)
);

CREATE TABLE release_phases (
    phase_id INTEGER PRIMARY KEY,
    release_id INTEGER REFERENCES releases(
      ↳release_id) ON DELETE CASCADE,
    name VARCHAR(50) NOT NULL, -- e.g., "Phase1
      ↳", "Phase2", "Phase3", "Phase4"
    sequence_number INTEGER NOT NULL, --
      ↳Determines the order of phases
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    created_by BIGINT REFERENCES users(user_id)
      ↳,
    updated_by BIGINT REFERENCES users(user_id)
      ↳,
    UNIQUE (release_id, name),
```

```
    UNIQUE (release_id, sequence_number)
);
```

Listing 5.6: Release and Phase Table Implementation

This implementation supports the bi-annual release cycle (e.g., "24.1", "24.3") used in automotive development, with each release progressing through four sequential phases. The `sequence_number` field provides explicit ordering of phases within a release, enabling efficient filtering and sorting in queries. Unique constraints ensure consistency of phase naming and sequencing within each release.

The ECU-phase mapping implements the association between ECUs and specific release phases:

```
CREATE TABLE ecu_phases (
    ecu_id INTEGER REFERENCES ecus(ecu_id) ON
      ↳DELETE CASCADE,
    phase_id INTEGER REFERENCES release_phases(
      ↳phase_id) ON DELETE CASCADE,
    release_type VARCHAR(50), -- e.g., "
      ↳Production", "Development", "Test"
    start_date DATE,
    end_date DATE,
    is_active BOOLEAN DEFAULT true,
    is_frozen BOOLEAN DEFAULT false,
    frozen_at TIMESTAMP WITH TIME ZONE,
    frozen_by BIGINT REFERENCES users(user_id),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    created_by BIGINT REFERENCES users(user_id)
      ↳,
    updated_by BIGINT REFERENCES users(user_id)
      ↳,
    PRIMARY KEY (ecu_id, phase_id),
    CONSTRAINT valid_date_range CHECK (
      ↳start_date <= end_date OR end_date IS
      ↳NULL)
);
```

Listing 5.7: ECU-Phase Mapping Implementation

This implementation extends beyond a simple junction table, incorporating status flags and temporal attributes tracking ECU progression through phases. The `is_frozen` flag implements the phase freezing mechanism, marking a phase as read-only when reaching a stable milestone. The ECU-phase mapping provides several capabilities supporting the versioning approach: independent progression of ECUs through development, explicit phase state tracking, and a fundamental versioning dimension associating parameters with specific development timeline points.

This represents a departure from traditional temporal database approaches. Rather than using validity timestamps and complex temporal queries, the system uses explicit phase associations to create a more intuitive and efficient versioning model. As noted by Bhattacherjee et al. [3], domain-specific versioning approaches often provide better performance and usability than generic temporal database techniques when tailored to specific application domain requirements.

5.1.4 Variant and Segment Management

The variant and segment management realizes the core parameter customization capabilities:

```
CREATE TABLE variants (
    variant_id BIGINT PRIMARY KEY,
    pid_id BIGINT REFERENCES pids(pid_id) ON
      ↳DELETE CASCADE,
    ecu_id INTEGER,
    phase_id INTEGER,
    name VARCHAR(100) NOT NULL,
    code_rule TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳ CURRENT_TIMESTAMP,
    created_by BIGINT REFERENCES users(user_id)
      ↳,
    updated_by BIGINT REFERENCES users(user_id)
      ↳,
```

```
    FOREIGN KEY (ecu_id, phase_id) REFERENCES ecu_phases(ecu_id, phase_id)
);

CREATE TABLE segments (
    segment_id BIGINT PRIMARY KEY,
    variant_id BIGINT REFERENCES variants(variant_id) ON DELETE CASCADE,
    parameter_id BIGINT REFERENCES parameters(parameter_id) ON DELETE CASCADE,
    dimension_index INTEGER NOT NULL,
    decimal NUMERIC NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    created_by BIGINT REFERENCES users(user_id),
    updated_by BIGINT REFERENCES users(user_id)
);
```

Listing 5.8: Variant and Segment Implementation

Each variant is associated with a specific PID, ECU, and phase, creating a three-way relationship placing the variant in both functional and temporal contexts. The `code_rule` field stores boolean expressions determining when a variant applies based on vehicle configuration codes, following the rule engine pattern described by Fowler [13].

Each segment associates a parameter with a variant, creating the core relationship defining parameter customization. The `dimension_index` field supports multi-dimensional parameters, allowing modification of specific elements within array parameters. Segments store values in a canonical decimal representation regardless of the parameter's native data type, implementing the canonical model pattern described by Hohpe and Woolf [15].

To support documentation and compliance requirements, the system implements a snapshot mechanism capturing variant and segment states at specific time points:

```
CREATE TABLE documentation_snapshots (
    snapshot_id INTEGER PRIMARY KEY,
```

```
    name VARCHAR(255) NOT NULL,
    description TEXT,
    ecu_id INTEGER,
    phase_id INTEGER,
    variant_count INTEGER DEFAULT 0,
    segment_count INTEGER DEFAULT 0,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    created_by BIGINT REFERENCES users(user_id),
    FOREIGN KEY (ecu_id, phase_id) REFERENCES ecu_phases(ecu_id, phase_id)
);

CREATE TABLE snapshot_variants (
    snapshot_variant_id INTEGER PRIMARY KEY,
    snapshot_id INTEGER REFERENCES documentation_snapshots(snapshot_id) ON DELETE CASCADE,
    original_variant_id BIGINT, -- Reference to the original variant
    pid_id BIGINT REFERENCES pids(pid_id) ON DELETE CASCADE,
    name VARCHAR(100) NOT NULL,
    code_rule TEXT,
    created_at TIMESTAMP WITH TIME ZONE,
    created_by BIGINT REFERENCES users(user_id)
);

CREATE TABLE snapshot_segments (
    snapshot_segment_id INTEGER PRIMARY KEY,
    snapshot_id INTEGER REFERENCES documentation_snapshots(snapshot_id) ON DELETE CASCADE,
    snapshot_variant_id INTEGER REFERENCES snapshot_variants(snapshot_variant_id) ON DELETE CASCADE,
    original_segment_id BIGINT, -- Reference to the original segment
    parameter_id BIGINT REFERENCES parameters(
```

```
    ↳parameter_id) ON DELETE CASCADE,
    dimension_index INTEGER NOT NULL,
    decimal NUMERIC NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE,
    created_by BIGINT REFERENCES users(user_id)
);
```

Listing 5.9: Documentation Snapshot Implementation

This implementation follows the snapshot pattern described by Fowler [13], creating a complete copy of variant and segment data at specific points. Rather than using a temporal database approach with validity periods, the system explicitly materializes historical states, ensuring they remain accessible regardless of subsequent modifications to live data. References to original entities enable traceability between snapshot and live data, implementing the origin tracking pattern described by Tichy [**?**].

In the automotive industry, these snapshots serve multiple purposes: providing immutable records of parameter configurations at significant development milestones, supporting quality assurance processes and regulatory compliance requirements, and facilitating comparative analysis between development phases.

5.1.5 Change Tracking Mechanisms

Comprehensive change tracking is implemented through the change history table, which records all modifications to key entities. Each change record includes entity type and ID, user, timestamp, change type, and both before and after states stored as JSONB documents. This approach implements the state snapshot pattern described by Fowler [13], capturing complete entity state rather than just modified fields.

The change tracking system incorporates a transaction ID to group related changes, implementing the unit of work pattern [13]. This grouping preserves the integrity of multi-entity operations, ensuring changes are interpreted in their proper context. The system serves multiple purposes beyond auditing: providing foundation for change history analysis, enabling detailed examination of parameter configuration evolution, and supporting accountability in the development process.

5.2 Version Control Implementation

The version control implementation constitutes the core VMAP system functionality, enabling parameter evolution management across different development phases.

5.2.1 Release Phase Management

The release phase management provides the temporal framework for parameter versioning. A phase relationship view establishes explicit navigation paths between phases in the development sequence:

```
CREATE OR REPLACE VIEW vw_phase_relationships AS
SELECT
    r.release_id,
    r.name AS release_name,
    rp.phase_id,
    rp.name AS phase_name,
    rp.sequence_number,
    (
        SELECT rp2.phase_id
        FROM release_phases rp2
        WHERE rp2.release_id = r.release_id
        AND rp2.sequence_number = rp.sequence_number + 1
    ) AS next_phase_id,
    (
        SELECT r2.release_id
        FROM releases r2
        WHERE r2.name > r.name
        ORDER BY r2.name
        LIMIT 1
    ) AS next_release_id
FROM
    releases r
    JOIN release_phases rp ON r.release_id = rp.release_id
WHERE
```

```
    r.is_active = true
ORDER BY
    r.name, rp.sequence_number;
```

Listing 5.10: Phase Relationship View Implementation

This view implements sophisticated temporal mapping, identifying both the next phase within a release and the initial phase of the next release. The implementation uses nested subqueries to determine sequential relationships, creating explicit links between phases across the development timeline. This follows the explicit relationship pattern described by Date [10], making temporal relationships queryable through standard SQL rather than relying on application logic to interpret sequence numbers.

Phase state management is implemented through stored procedures that control phase transitions between editable and frozen states. The freeze operation records the user, timestamp, and reason, maintaining accountability for these significant state changes. Additionally, a documentation snapshot is automatically created to preserve the complete parameter configuration at freezing time.

5.2.2 Parameter History Tracking

Parameter history tracking extends the core change tracking mechanism with specific features for analyzing parameter evolution over time:

```
CREATE OR REPLACE FUNCTION
  ↳get_parameter_history(
    p_parameter_id BIGINT,
    p_dimension_index INTEGER DEFAULT NULL
)
RETURNS TABLE (
    change_id BIGINT,
    changed_at TIMESTAMP WITH TIME ZONE,
    user_name VARCHAR,
    variant_id BIGINT,
    variant_name VARCHAR,
    old_value NUMERIC,
    new_value NUMERIC,
    change_type VARCHAR,
    transaction_id BIGINT
```

```sql
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        ch.change_id,
        ch.changed_at,
        u.first_name || ' ' || u.last_name AS user_name,
        v.variant_id,
        v.name AS variant_name,
        CASE
            WHEN ch.change_type = 'DELETE' THEN
                (ch.old_values->>'decimal')::NUMERIC
            WHEN ch.change_type = 'UPDATE' THEN
                (ch.old_values->>'decimal')::NUMERIC
            ELSE NULL
        END AS old_value,
        CASE
            WHEN ch.change_type = 'CREATE' THEN
                (ch.new_values->>'decimal')::NUMERIC
            WHEN ch.change_type = 'UPDATE' THEN
                (ch.new_values->>'decimal')::NUMERIC
            ELSE NULL
        END AS new_value,
        ch.change_type,
        ch.transaction_id
    FROM
        change_history ch
        JOIN users u ON ch.user_id = u.user_id
        JOIN segments s ON ch.entity_id = s.segment_id
                    AND ch.entity_type = 'segments'
        JOIN variants v ON s.variant_id = v.variant_id
    WHERE
```

```
            s.parameter_id = p_parameter_id
            AND (p_dimension_index IS NULL
                 OR s.dimension_index =
                    ↳p_dimension_index)
        ORDER BY
            ch.changed_at DESC;
END;
$$ LANGUAGE plpgsql;
```

Listing 5.11: Parameter History Function Implementation

This function uses JSON path operators to extract and type-cast values from the JSONB store, implementing the document extraction pattern described by Abelló et al. [?]. It handles different data structures for different change types while presenting a consistent interface to the caller. By including user information in the result set, it implements the attribution pattern described by Fowler [13], providing immediate context for who made each change without requiring additional queries.

5.2.3 Freeze Mechanism Implementation

The freeze mechanism is enforced through triggers that prevent modifications to frozen phases:

```
CREATE OR REPLACE FUNCTION
   ↳enforce_freeze_status()
RETURNS TRIGGER AS $$
DECLARE
    is_frozen BOOLEAN;
BEGIN
    -- Get freeze status for the relevant phase
    IF TG_TABLE_NAME = 'variants' THEN
        SELECT ep.is_frozen INTO is_frozen
        FROM ecu_phases ep
        WHERE ep.ecu_id = NEW.ecu_id AND ep.
           ↳phase_id = NEW.phase_id;
    ELSIF TG_TABLE_NAME = 'segments' THEN
        SELECT ep.is_frozen INTO is_frozen
        FROM variants v
```

```
            JOIN ecu_phases ep ON v.ecu_id = ep.ecu_id
                                 AND v.phase_id = ep.phase_id
            WHERE v.variant_id = NEW.variant_id;
        END IF;

        -- Prevent modifications if phase is frozen
        IF is_frozen THEN
            RAISE EXCEPTION 'Cannot modify % in frozen phase', TG_TABLE_NAME;
        END IF;

        RETURN NEW;
    END;
    $$ LANGUAGE plpgsql;
```

Listing 5.12: Freeze Status Enforcement Implementation

This implementation uses a polymorphic trigger approach working with multiple entity types, implementing the shared constraint pattern described by Karwin [18]. It includes conditional logic navigating different relationship paths depending on entity type. The explicit exceptions with descriptive messages implement the informative error pattern described by Molinaro [24], providing clear feedback when modifications are attempted on frozen phases.

5.3 Query Optimization Strategies

The VMAP system implements sophisticated query optimization strategies ensuring responsive performance despite the complexity of parameter data and version control requirements.

5.3.1 Indexing Strategy

A comprehensive indexing strategy accelerates common query patterns while balancing performance and storage requirements:

```sql
-- Core entity hierarchy indexes
CREATE INDEX idx_modules_ecu ON ecu_modules(ecu_id);
CREATE INDEX idx_pids_ecu_module ON pids(ecu_id, module_id);
CREATE INDEX idx_parameters_pid_phase ON parameters(pid_id, phase_id);
CREATE INDEX idx_variants_pid_phase ON variants(pid_id, phase_id);
CREATE INDEX idx_segments_variant ON segments(variant_id);
CREATE INDEX idx_segments_parameter ON segments(parameter_id);

-- Covering indexes for common joins
CREATE INDEX idx_variants_complete ON variants(variant_id, pid_id,
                                               ecu_id, phase_id, name);
CREATE INDEX idx_parameters_complete ON parameters(parameter_id, pid_id,
                                                   name, ecu_id, phase_id);
```

Listing 5.13: Hierarchical Access Path Indexing Implementation

These indexes match the natural navigation paths in the data model, implementing the access path optimization pattern described by Molinaro [24]. The covering indexes contain all columns needed for common operations, implementing the index-only scan pattern described by Karwin [18].

Text search optimization uses PostgreSQL's trigram indexing extension, enabling efficient partial-match and similarity-based searches. According to Obe and Hsu [26], these indexes accelerate pattern matching operations for parameter names and identifiers:

```
-- Create the pg_trgm extension for similarity searching
CREATE EXTENSION IF NOT EXISTS pg_trgm;

-- Trigram indexes for name searching
CREATE INDEX idx_parameters_name_trgm
    ON parameters USING gin (name gin_trgm_ops);
CREATE INDEX idx_parameters_parameter_name_trgm
    ON parameters USING gin (parameter_name gin_trgm_ops);
CREATE INDEX idx_variants_name_trgm
    ON variants USING gin (name gin_trgm_ops);
```

Listing 5.14: Text Search Optimization Implementation

These domain-specific indexing strategies reflect the unique requirements of automotive parameter management, where efficient navigation through complex hierarchical and temporal structures is essential for system responsiveness. The implemented strategy achieves balance by focusing on the most common access patterns while avoiding redundant or rarely-used indexes.

5.3.2 Partitioning Implementation

For large-scale implementations with extensive history tracking, table partitioning improves performance and manageability:

```
-- Partitioned change history table
CREATE TABLE change_history_partitioned (
    change_id BIGINT NOT NULL,
    user_id BIGINT,
    ecu_id INTEGER,
    phase_id INTEGER,
    entity_type VARCHAR(50) NOT NULL,
    entity_id BIGINT NOT NULL,
    change_type VARCHAR(50),
    old_values JSONB,
    new_values JSONB,
```

```
    changed_at TIMESTAMP WITH TIME ZONE NOT
      NULL,
    transaction_id BIGINT NOT NULL
) PARTITION BY RANGE (changed_at);

-- Create monthly partitions for change history
CREATE TABLE change_history_y2024m01 PARTITION
  OF change_history_partitioned
    FOR VALUES FROM ('2024-01-01') TO ('
      2024-02-01');
```

Listing 5.15: Table Partitioning Implementation

This approach uses PostgreSQL's declarative partitioning feature, implementing the range partitioning pattern described by Obe and Hsu [26]. According to Bhattacherjee et al. [3], partitioning large historical tables provides several benefits: improved query performance for time-bounded queries, efficient archiving of old partitions to slower storage, simplified maintenance operations, and better parallelization of queries across partitions.

The partitioning strategy specifically targets the change history table because of its continuous growth characteristics. By partitioning this table by month, the implementation achieves the performance benefits noted by Elmasri and Navathe [11] for time-series data management, where queries typically focus on specific time periods rather than the entire history.

5.4 Integration Implementation

The integration implementation connects VMAP with external enterprise systems providing parameter definitions and vehicle configuration data. The implementation focuses on two primary integration points: the Parameter Definition Database (PDD) and the Vehicle Configuration Database (VCD).

5.4.1 Parameter Definition Database Synchronization

The database infrastructure for PDD synchronization includes tables tracking synchronization operations and results:

```sql
CREATE TABLE pdd_databases (
    database_id BIGINT PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    last_updated_at TIMESTAMP WITH TIME ZONE NOT NULL,
    description TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE pdd_sync_history (
    sync_id INTEGER PRIMARY KEY,
    ecu_id INTEGER,
    phase_id INTEGER,
    sync_date TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(50) NOT NULL,
    modules_count INTEGER DEFAULT 0,
    pids_count INTEGER DEFAULT 0,
    parameters_count INTEGER DEFAULT 0,
    entity_changes TEXT,
    executed_by BIGINT REFERENCES users(user_id),
    FOREIGN KEY (ecu_id, phase_id) REFERENCES ecu_phases(ecu_id, phase_id)
);
```

Listing 5.16: Parameter Definition Database Synchronization Infrastructure

This infrastructure implements the external system registry pattern described by Hohpe and Woolf [15], tracking available Parameter Definition Databases and their last update times. The detailed synchronization operations history implements the integration audit pattern, capturing what entities were affected and enabling traceability and data flow analysis between systems.

The phase-based synchronization approach was selected after careful analysis of involved trade-offs. This approach simplifies query patterns by establishing direct relationships between parameters and release phases, enabling efficient retrieval without complex version reconstruction. It aligns naturally with the automotive development process, where each release phase represents a distinct development milestone, making the system more intuitive for domain experts [35].

5.4.2 Vehicle Configuration Database Exchange

The database structure for vehicle configuration data includes tables for vehicles and associated codes:

```sql
CREATE TABLE vcd_vehicles (
    vehicle_id INTEGER PRIMARY KEY,
    vcd_vehicle_id VARCHAR(100) UNIQUE NOT NULL,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    is_active BOOLEAN DEFAULT true,
    last_sync_at TIMESTAMP WITHOUT TIME ZONE,
    created_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE vehicle_codes (
    code_id INTEGER PRIMARY KEY,
    vcd_code_id VARCHAR(100) UNIQUE,
    code VARCHAR(50) NOT NULL,
    vehicle_type VARCHAR(100),
    description TEXT,
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE vehicle_code_mapping (
```

```
        vehicle_id INTEGER REFERENCES
          ↳vcd_vehicles(vehicle_id)
                    ON DELETE CASCADE,
        code_id INTEGER REFERENCES
          ↳vehicle_codes(code_id)
                  ON DELETE CASCADE,
        created_at TIMESTAMP WITHOUT TIME ZONE
          ↳DEFAULT CURRENT_TIMESTAMP,
        PRIMARY KEY (vehicle_id, code_id)
    );
```

Listing 5.17: Vehicle Configuration Storage

This structure separates vehicles from codes with a mapping table, implementing the many-to-many relationship pattern described by Elmasri and Navathe [11]. External identifiers (`vcd_vehicle_id`, `vcd_code_id`) link to the Vehicle Configuration Database system, implementing the integration reference pattern described by Hohpe and Woolf [15].

A critical component is the code rule evaluation engine, which determines when variants apply to specific vehicles. This engine implements a stack-based parsing approach to evaluate boolean expressions, handling complex boolean logic used in variant code rules. According to Molinaro [24], this approach provides an efficient mechanism for processing complex conditions involving multiple operators and precedence rules.

The vehicle data synchronization uses a JSON-based data transfer approach, implementing the document exchange pattern described by Kleppmann and Beresford [20]. UPSERT operations ensure idempotent synchronization that can be run multiple times without creating duplicate entities, supporting both initial loading and incremental updates.

5.4.3 Parameter File Generation Support

The parameter file generation capability represents a core integration point between the version control system and vehicle testing infrastructure, transforming abstract parameter configurations into concrete binary files that can be loaded onto electronic control units:

```
    CREATE TABLE par_files (
        par_file_id BIGINT PRIMARY KEY,
```

```
        vehicle_id INTEGER REFERENCES
          ↳vcd_vehicles(vehicle_id),
        ecu_id INTEGER,
        phase_id INTEGER,
        generated_at TIMESTAMP WITH TIME ZONE
          ↳DEFAULT CURRENT_TIMESTAMP,
        generated_by BIGINT REFERENCES users(
          ↳user_id),
        description TEXT,
        FOREIGN KEY (ecu_id, phase_id)
          ↳REFERENCES ecu_phases(ecu_id,
          ↳phase_id)
    );
```

Listing 5.18: Parameter File Generation Records

The parameter file generation process integrates version control capabilities with both vehicle configuration data and parameter definitions to produce vehicle-specific parameter configurations. According to Staron [35], this integration point is critical in automotive software development, transforming abstract parameter configurations into testable implementations that can be validated on actual hardware.

The parameter file generation algorithm applies a sophisticated variant resolution process that evaluates code rules against vehicle configurations to determine which variants apply to a specific vehicle. This process implements the rule evaluation pattern described by Fowler [13], applying a consistent rule evaluation algorithm to determine applicable parameter values. According to Broy [6], this rule-based configuration approach is essential in automotive software development, where vehicles may have thousands of possible configurations that must be handled systematically.

5.4.4 Integration Architecture Overview

The integration architecture follows an enterprise integration pattern approach with clear separation between the core versioning system and external data sources. This separation is implemented through specialized data structures mapping between external identifiers and internal entities, enabling the system to maintain stable internal references even as external systems evolve.

The asynchronous integration model allows VMAP to continue operating even when external systems are unavailable, implementing the loose coupling pattern described by Hohpe and Woolf [15]. According to Trovão [38], this decoupling is particularly important in automotive development environments, where multiple systems from different vendors must collaborate without tight dependencies.

The database architecture includes explicit tracking of integration operations, implementing the integration audit pattern. This comprehensive logging enables troubleshooting of integration issues, analysis of data flows between systems, and verification of synchronization completeness. According to Bhattacherjee et al. [3], this audit capability is essential for maintaining data integrity in integrated systems, where inconsistencies between systems can lead to significant quality issues.

5.5 Security Implementation

The security implementation addresses critical requirements for protecting parameter data and enforcing appropriate access controls, ensuring users can only perform actions appropriate to their roles and responsibilities.

5.5.1 Role-Based Access Control

The core of the RBAC system is the permission verification function:

```
CREATE OR REPLACE FUNCTION has_permission(
    p_user_id BIGINT,
    p_permission_name VARCHAR
)
RETURNS BOOLEAN AS $$
DECLARE
    v_has_permission BOOLEAN;
BEGIN
    -- Check if user has the permission
      ↳through any role
    SELECT EXISTS (
        SELECT 1
        FROM user_roles ur
        JOIN role_permissions rp ON ur.
          ↳role_id = rp.role_id
```

```sql
            JOIN permissions p ON rp.permission_id = p.permission_id
            WHERE ur.user_id = p_user_id
            AND p.name = p_permission_name
        ) INTO v_has_permission;

        -- If not found in roles, check direct user permissions
        IF NOT v_has_permission THEN
            SELECT EXISTS (
                SELECT 1
                FROM user_permissions up
                JOIN permissions p ON up.permission_id = p.permission_id
                WHERE up.user_id = p_user_id
                AND p.name = p_permission_name
            ) INTO v_has_permission;
        END IF;

        RETURN v_has_permission;
    END;
    $$ LANGUAGE plpgsql;
```

Listing 5.19: Permission Verification Function

This function implements a hybrid approach checking for permissions granted through roles and then falling back to direct user permissions if necessary. According to Ferraiolo et al. [12], this dual-check approach provides flexibility for exceptional cases while maintaining the structural benefits of role-based access control. The implementation uses efficient EXISTS queries rather than retrieving complete permission records, optimizing performance for this high-frequency operation.

Beyond basic permission checking, the system implements module-specific access control verification, combining role-based and module-based access control in a unified framework. This approach follows the principle of least privilege, allowing precise control over which modules a user can modify. Sandhu and Bhamidipati [29] note that this layered authorization approach is particularly valuable in complex development environments where responsibilities are divided among specialized teams.

5.5.2 Data Security Measures

The implementation includes several key data security measures protecting parameter configuration integrity. The freeze mechanism implements strict enforcement of the read-only state for release phases that have reached development milestones. Database triggers enforce phase state restrictions at the database level, ensuring protection cannot be bypassed by application code. According to Date [10], this constraint enforcement approach provides a fundamental layer of data security operating independently of application logic.

Parameter value validation enforces parameter-specific constraints, ensuring values remain within physically meaningful and safe ranges. This preventive validation pattern described by Karwin [18] prevents invalid data from being stored in the database, maintaining data integrity throughout the system. Molinaro [24] notes that this approach to data validation is particularly important in safety-critical systems where incorrect parameter values could lead to system malfunctions.

5.5.3 Audit Trail Implementation

The audit trail implementation provides comprehensive tracking of all parameter data changes, supporting accountability and compliance requirements. This capability is particularly important in automotive software development, where regulatory frameworks often require detailed traceability of all configuration changes.

Automatic change tracking is implemented through database triggers:

```
    CREATE OR REPLACE FUNCTION log_change()
RETURNS TRIGGER AS $
    DECLARE
        transaction_id BIGINT;
        change_type VARCHAR(50);
        old_values JSONB;
        new_values JSONB;
        entity_id BIGINT;
        phase_id INTEGER;
        ecu_id INTEGER;
    BEGIN
        -- Get the current transaction ID or
          ↳create a new one
```

```
        SELECT COALESCE(NULLIF(current_setting('app.transaction_id', true), ''),
                        nextval('change_history_transaction_id_s'))
        INTO transaction_id;

    -- Determine entity_id and other values based on operation
    -- [implementation details omitted for brevity]

        -- Capture entity states
        IF TG_OP = 'INSERT' THEN
            change_type := 'CREATE';
            old_values := NULL;
            new_values := to_jsonb(NEW);
        ELSIF TG_OP = 'UPDATE' THEN
            change_type := 'UPDATE';
            old_values := to_jsonb(OLD);
            new_values := to_jsonb(NEW);
        ELSIF TG_OP = 'DELETE' THEN
            change_type := 'DELETE';
            old_values := to_jsonb(OLD);
            new_values := NULL;
        END IF;

        -- Insert into change_history
        INSERT INTO change_history (
            user_id, ecu_id, phase_id, entity_type, entity_id,
            change_type, old_values, new_values, transaction_id
        ) VALUES (
            NULLIF(current_setting('app.user_id', true), '')::BIGINT,
            ecu_id, phase_id, TG_TABLE_NAME, entity_id,
            change_type, old_values, new_values, transaction_id
```

```
            );

            RETURN NULL;
        END;
$ LANGUAGE plpgsql;
```

Listing 5.20: Change History Trigger

This implementation applies change logging to all critical entities, implementing the universal auditing pattern described by Fowler [13]. It captures complete entity state rather than just modified fields, implementing the state snapshot pattern. By using JSONB for storing entity states, the implementation provides flexibility to accommodate different entity structures while enabling efficient querying of change details using PostgreSQL's JSON operators.

The audit trail includes sophisticated analysis capabilities through specialized functions retrieving entity history and transaction changes. These functions implement the entity timeline pattern described by Fowler [13] and the transaction context pattern described by Date [10], providing both detailed and contextual views of system changes. According to Staron [35], these comprehensive audit capabilities are essential for meeting regulatory compliance requirements in automotive software development, where parameter changes must be traceable throughout the development lifecycle.

6 Evaluation and Validation

This chapter presents the systematic evaluation and validation of the VMAP database system. Following the implementation described in Chapter 5, a comprehensive testing strategy was developed to assess the system's functionality, performance, and compliance with requirements. The evaluation process focused on four key areas: user management, release management, parameter versioning, and variant management, using both controlled test scenarios and production-scale data volumes. Rather than an exhaustive documentation of all tests, this chapter highlights representative test cases and key findings that demonstrate the system's capabilities and limitations.

6.1 Validation Methodology

The validation methodology followed a structured approach combining functional testing, performance analysis, and integration verification. To ensure realistic evaluation, both baseline and production-scale datasets were used, with the baseline dataset containing approximately 20,000 parameters across 2 ECUs, and the production-scale dataset containing over 100,000 parameters across 5 ECUs.

6.1.1 Test Scenario Development

Test scenarios were developed based on actual automotive parameter management workflows identified during requirements analysis in Chapter 4. Each test scenario was designed to validate specific functional requirements while reflecting real-world usage patterns. The scenarios incorporated representative tasks for each user role and followed complete workflow sequences from parameter definition through variant creation to documentation.

The test scenarios were categorized into functional areas corresponding to the primary system capabilities:

$$\begin{aligned}
\text{User Management} &: \text{Authentication, authorization, role assignment, module access}\\
\text{Release Management} &: \text{Phase transitions, freeze operations}\\
\text{Variant Management} &: \text{Variant creation, segment modification}\\
\text{Integration} &: \text{PDD synchronization, vehicle configuration}
\end{aligned}$$

Each scenario was implemented as a structured test case with defined inputs, expected outcomes, and verification steps at both the application and database levels. The test design followed a modified version of Molinaro's approach to database validation [24], with additional emphasis on traceability between requirements and test cases.

6.1.2 Performance Measurement Framework

A performance measurement framework was established to assess system responsiveness and resource utilization under various operational conditions. Key performance indicators were defined based on system requirements, including query response time, transaction throughput, database size growth patterns, memory utilization, and execution time for batch operations.

Performance measurements were conducted on a standardized test environment matching the target production specifications: PostgreSQL 17 running on a server with 8 vCPUs, 32GB RAM, and SSD storage. All tests were performed with both the baseline dataset and the production-scale dataset to assess scaling characteristics.

The measurement methodology employed automated test scripts with integrated timing capture, following the principles outlined by Zaitsev et al. [31] for database performance evaluation. Each test was executed multiple times with results averaged to account for system variations, and outliers were identified and analyzed for potential optimization opportunities.

6.2 Functional Testing Results

Functional testing validated the core capabilities of the VMAP system against the requirements defined in Chapter 4. This section presents the key findings for each functional area, focusing on representative test cases and critical system behaviors.

6.2.1 User Management Validation

The user management and access control system was evaluated through a focused testing approach to verify the implementation of the hybrid role-permission model described in Section 4.1.3. The validation methodology applied a structured approach with distinct verification techniques including functional permission verification, access boundary testing, permission inheritance validation, and cross-role security verification.

As Sandhu et al. [30] emphasize, effective evaluation of role-based access control requires testing both positive permissions (granted access) and negative permissions (denied access) across role boundaries.

A comprehensive yet efficient test matrix was developed encompassing 42 distinct test scenarios strategically distributed across four verification domains: role-based permissions, module-based access control, direct permission assignment, and phase-specific permissions. Each test case verified a specific permission boundary with separate validation at both service and database layers. This focused approach aligns with Molinaro's principles for database validation [24], which emphasizes targeted verification of critical constraints over exhaustive testing.

Table A.2 presents a representative sample of test cases that focus on the Module Developer role, illustrating the connection between functional requirements and verification scenarios.

Table 6.1: Sample Module Developer Role Permission Test Cases

ID	Description	Test Action	Expected Outcome	Status
MD-01	Create Variant (Assigned Module)	Create new variant for parameter in assigned module	Variant created successfully	Pass
MD-02	Create Variant (Unassigned Module)	Create new variant for parameter in unassigned module	Access denied error	Pass
MD-03	Edit Variant (Assigned Module)	Modify existing variant code rule	Variant updated successfully	Pass
MD-04	Delete Variant	Attempt to delete variant	Access denied error	Pass
MD-05	Create Segment (Assigned Module)	Create new segment with valid value	Segment created successfully	Pass
MD-06	Modify Frozen Phase	Attempt to modify segment in frozen phase	Access denied error	Pass

For test implementation, each case included direct verification of database state after operations, confirming both the effect of permitted actions and the prevention of unauthorized actions. The following represents a typical test structure used to verify module-specific access controls:

```
// Scenario: Module Developer attempting to create variant in unassigned module
// Arrange: Set up test user and unassigned module parameter
var user = GetTestUser("module_developer@example.com");
var unassignedParameter = GetParameterFromUnassignedModule();
var variant = CreateVariantForParameter(unassignedParameter);

// Act & Assert: Verify permission is denied
var exception = Assert.Throws<PermissionDeniedException>(() =>
    _variantService.CreateVariant(variant, user.UserId));
Assert.That(exception.Message, Contains.Substring("No write access"));

// Verify no database change occurred
var dbVariant = _database.QuerySingleOrDefault<Variant>(
    "SELECT * FROM variants WHERE name = @Name",
    new { Name = variant.Name });
Assert.IsNull(dbVariant);
```

Listing 6.1: Representative Test Case Structure

The module-based access control tests were particularly critical as they represent a departure from standard RBAC patterns as defined by Sandhu et al. [30], implementing instead a hybrid attribute-enhanced approach similar to that described by Kuhn et al. [21]. All ten module assignment validation tests passed successfully, confirming compliance with Ferraiolo et al.'s [12] recommendations for combining role-based and attribute-based access control models. The tests verified that write access was correctly limited to assigned modules for Module Developers while read access remained available for all modules, implementing the principle of least privilege as recommended by Sandhu and Bhamidipati [29].

Direct permission assignment tests confirmed that user-specific permissions effectively overrode role defaults in all test scenarios. This capability is essential for supporting exception cases in complex organizational structures as noted by Hu et al. [16]. The six test cases targeting this area verified both the granting of additional permissions and the removal of permissions that would normally be inherited through role assignments.

Phase-specific permission tests validated the interaction between the access control system and the phase management framework. All six test cases in this domain passed successfully, confirming that modifications to frozen phases were properly prevented while still allowing appropriate access for documentation purposes. This validation addresses a critical requirement for regulated development processes as described by Staron [35], where development milestone integrity must be preserved. Table 6.2 summarizes the key findings from user management testing across different test categories.

Table 6.2: User Management Test Results

Test Category	Results
Role Permission Validation	All permissions correctly applied through roles (16/16 test cases)
Module-Based Access	Write access correctly limited to assigned modules (10/10 test cases)
Direct Permission Assignment	User-specific permissions overrode role defaults (6/6 test cases)
Phase-Specific Permissions	Frozen phase protection enforced correctly (6/6 test cases)
Boundary Cases	Edge conditions handled appropriately (4/4 test cases)

The complete set of test cases is documented in Appendix A, covering all access control aspects across different user roles, module assignments, and system states.

Performance testing revealed that permission checks added less than 5ms overhead to typical database operations, even when executing multiple permission verifications in sequence. This aligns with Hu et al.'s [16] recommendations for optimized permission validation in enterprise systems. The performance was achieved through strategic denormalization and view-based permission aggregation as described in Section **??**.

The audit trail verification confirmed that all security-related operations were properly logged with complete metadata, including the user making the change, timestamp, and specific permissions affected. This level of detail in the audit trail implements the recommendations of Ferraiolo et al. [12] for maintaining accountability in security-sensitive operations. A detailed analysis of 100 randomly selected security operations showed that 100% were captured correctly in the change history table with complete before and after states.

6.2.2 Release Management Validation

Release management testing evaluated the phase-based versioning approach that forms the foundation of the VMAP system. Testing focused on four key aspects: phase sequence validation, phase transition operations, freeze functionality, and phase comparison.

Phase sequence validation confirmed that the system correctly enforced the defined sequence of development phases (Phase1 → Phase2 → Phase3 → Phase4) with successful validation across all 24 test cases. This sequential enforcement is essential for maintaining the structured development workflow described by Broy [6] for automotive software development.

Phase transition testing verified that parameter configurations were correctly copied between phases with complete preservation of parameter-variant-segment relationships. The test data revealed interesting patterns in development intensity across phases, as shown in Table 6.3.

Table 6.3: Phase Transition Test Results

Transition Type	Variants	Segments	Added Variants	Added Segments	Time
Baseline Dataset					
Phase1	188	28,776	-	-	-
Phase1 → Phase2	188	28,776	90	14,104	2.51s
Phase2 → Phase3	278	42,880	0	0	2.96s
Phase3 → Phase4	278	42,880	0	0	2.97s
Full Dataset					
Phase1	830	167,990	-	-	-
Phase1 → Phase2	830	167,990	170	41,113	12.39s
Phase2 → Phase3	1,000	209,103	0	0	12.87s
Phase3 → Phase4	1,000	209,103	0	0	12.89

The test results reveal a significant pattern in development intensity across phases, consistent with Staron's observations [35] regarding automotive software development cycles. The data demonstrates that the majority of parameter configurations occur during Phase1, with substantial additions in Phase2. In contrast, Phase3 and Phase4 typically involve refinement and validation rather than introducing new parameters or variants. This concentration of development activity in early phases aligns with the V-model approach common in automotive software development [27], where early phases focus on implementation while later phases emphasize validation and verification.

Phase transition performance characteristics showed only modest increases in execution time despite growing data volumes across phases. For the baseline dataset, transition times increased from 2.51s for Phase1→Phase2 to 2.96s for Phase2→Phase3 and 2.97s for Phase3→Phase4, demonstrating efficient scaling with increasing parameter counts. The slight increase in execution time between Phase2→Phase3 and Phase3→Phase4, despite no new variants or segments being added, suggests that total data volume remains the primary factor affecting transition performance. Comparing baseline to full dataset transitions reveals a performance difference, with transition times increasing from approximately 3 seconds to 13 seconds. This represents a sublinear scaling factor of approximately 4.3x for a dataset size increase of 5.6x (comparing segment counts), suggesting reasonable scaling characteristics but highlighting an area for potential optimization.

As noted by Trovão [38], later phases in automotive parameter development typically focus on refinement rather than wholesale changes, with modifications targeting specific parameters based on testing feedback. This pattern is reflected in the test data, which shows significant additions in early phases but no new variants or segments in Phase3 and Phase4. While the test data might suggest that phase transition time increases gradually as development progresses, in real-world scenarios, the creation of new variants and segments is not always proportional to phase progression. Instead, existing variants and segments are often updated based on phase-specific requirements without necessarily creating new entities. This observation supports the architectural decision to optimize the phase transition mechanism for selective propagation of changes rather than complete data replication.

Phase freezing functionality was validated through ten specific test cases targeting both database-level constraints and service-layer restrictions. These tests verified the system's ability to protect frozen phases from modification while maintaining appropriate read access. Table 6.9 details the specific test cases and their results.

Table 6.4: Phase Freeze Protection Test Cases

ID	Test Case	Test Action	Expected Outcome	Result
FRZ-01	Direct SQL INSERT on variants	Execute INSERT statement on frozen phase	Operation blocked with error message	Pass
FRZ-02	Direct SQL UPDATE on segments	Execute UPDATE statement on frozen phase	Operation blocked with error message	Pass
FRZ-03	Direct SQL DELETE on variants	Execute DELETE statement on frozen phase	Operation blocked with error message	Pass
FRZ-04	VariantService. CreateVariant()	Attempt to create variant in frozen phase	PhaseFreezed Exception thrown	Pass
FRZ-05	VariantService. UpdateVariant()	Attempt to update variant in frozen phase	PhaseFreezed Exception thrown	Pass
FRZ-06	SegmentService. CreateSegment()	Attempt to create segment in frozen phase	PhaseFreezed Exception thrown	Pass
FRZ-07	SegmentService. UpdateSegment()	Attempt to update segment in frozen phase	PhaseFreezed Exception thrown	Pass
FRZ-08	SegmentService. DeleteSegment()	Attempt to delete segment in frozen phase	PhaseFreezed Exception thrown	Pass
FRZ-09	DocumentationService. CreateSnapshot()	Create documentation snapshot of frozen phase	Snapshot created successfully	Pass
FRZ-10	ParFileService. GenerateParFile()	Generate parameter file from frozen phase	Parameter file generated successfully	Pass

For each write operation test case (FRZ-01 through FRZ-08), verification included both confirmation that the expected exception was thrown and that no database changes occurred, maintaining data integrity. The read operation test cases (FRZ-09 and FRZ-10) verified that read access remained available with minimal performance impact

(<5ms overhead). Across 150 test operations targeting frozen phases, the system successfully prevented all modification attempts while maintaining appropriate read access, implementing the controlled milestone management required for regulated development environments as described by Staron [35].

6.2.3 Variant Management Validation

Variant management validation focused on assessing the system's capabilities for handling parameter customization through variants and segments. Testing employed a methodical approach across two interrelated domains: variant creation and segment modification workflows. Each testing domain was evaluated using both the baseline dataset (188 variants, 28,776 segments) and the production-scale dataset (830 variants, 167,990 segments) to analyze functionality and performance under varying data volumes.

Variant creation validation encompassed 18 distinct test cases designed to exercise the full spectrum of variant operations. These tests verified proper implementation of domain constraints as defined in the conceptual architecture (Section **??**). For variant creation, test cases included validation of unique name constraints within PIDs, verification of proper code rule storage, and confirmation of correct relationship establishment between variants and their parent PIDs. All test cases passed successfully for both scalar and complex parameters, with constraint enforcement consistently preventing invalid operations. As Karwin [18] notes, constraint-based validation provides a robust foundation for maintaining data integrity in complex relational systems.

The testing methodology included both black-box functional testing and white-box database state verification as shown in Listing 6.2.

```sql
-- Verification query executed after variant
   ↳creation operations
SELECT
    v.variant_id, v.name, v.code_rule, v.
       ↳created_by, v.created_at,
    EXISTS (
        SELECT 1 FROM change_history ch
        WHERE ch.entity_type = 'variants'
        AND ch.entity_id = v.variant_id
        AND ch.change_type = 'CREATE'
    ) AS has_audit_trail
FROM
```

```
    variants v
WHERE
    v.pid_id = @test_pid_id
    AND v.phase_id = @test_phase_id
ORDER BY
    v.created_at DESC
LIMIT 1;
```

Listing 6.2: Variant Creation Verification Query

Audit trail analysis revealed comprehensive capture of variant operations, with 100% of test operations correctly recorded in the change history with complete metadata as shown in 6.1. The audit trail included proper attribution of each change to specific users, accurate timestamps, and complete before/after state capture for modified entities. This implementation aligns with Bhattacherjee's recommendations [3] for maintaining comprehensive provenance information in versioned datasets.

Performance analysis of variant operations revealed consistent response times across different variant complexities. Table 6.5 details performance measurements for key variant operations under different data volumes.

Table 6.5: Variant Operation Performance Metrics

Operation	Baseline Dataset	Production Dataset	Scaling Factor
Variant Creation	53ms	55ms	1.03x
Variant Update	86ms	124ms	1.44x
Variant Retrieval	45ms	72ms	1.60x
Variant Listing (per PID)	38ms	68ms	1.79x

The sublinear scaling characteristics observed in these measurements validate the effectiveness of the database schema design and indexing strategy described in Section **??**. As noted by Obe and Hsu [26], properly designed covering indexes significantly improve query performance for entity retrieval operations, particularly when filtering by composite attributes.

Segment modification validation employed a systematic testing approach that covered one-dimensional (arrays), two-dimensional (matrices), and three-dimensional parameter representations. Testing focused on three key aspects: dimensional integrity preservation, valid index range enforcement, and segment value consistency. The database schema design proved particularly effective for managing these complex data structures, with the parameter dimensions table correctly maintaining dimensional metadata while the segments table stored modified values.

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------------------------
--------------------------
change_id      | 201
user_id        | 3
ecu_id         | 1
phase_id       | 4
entity_type    | variants
entity_id      | 23
change_type    | CREATE
old_values     |
new_values     | {"name": "Var77", "ecu_id": 1, "pid_id": 3, "phase_id": 4, "code_rule": "AA7", "created_by": 1, "update
d_by": 1, "variant_id": 23}
changed_at     | 2025-04-06 09:44:30.60209+02
transaction_id | 113
-[ RECORD 2 ]--+-----------------------------------------------------------------------------------------------
--------------------------
change_id      | 202
user_id        | 1
ecu_id         | 1
phase_id       | 1
entity_type    | segments
entity_id      | 56
change_type    | CREATE
old_values     |
new_values     | {"decimal": 5, "created_by": 1, "segment_id": 56, "updated_by": 1, "variant_id": 10, "parameter_id": 69
, "dimension_index": 0}
changed_at     | 2025-04-06 10:03:37.158898+02
transaction_id | 114
-[ RECORD 3 ]--+-----------------------------------------------------------------------------------------------
--------------------------
change_id      | 203
user_id        | 1
ecu_id         | 1
phase_id       | 1
entity_type    | segments
entity_id      | 56
change_type    | UPDATE
old_values     | {"decimal": 5, "created_by": 1, "segment_id": 56, "updated_by": 1, "variant_id": 10, "parameter_id": 69
, "dimension_index": 0}
new_values     | {"decimal": 4, "created_by": 1, "segment_id": 56, "updated_by": 1, "variant_id": 10, "parameter_id": 69
, "dimension_index": 0}
changed_at     | 2025-04-06 10:03:41.714021+02
transaction_id | 115
-[ RECORD 4 ]--+-----------------------------------------------------------------------------------------------
--------------------------
change_id      | 204
user_id        | 5
ecu_id         | 1
phase_id       | 1
entity_type    | segments
entity_id      | 56
change_type    | DELETE
old_values     | {"decimal": 4, "created_by": 1, "segment_id": 56, "updated_by": 1, "variant_id": 10, "parameter_id": 69
, "dimension_index": 0}
new_values     |
changed_at     | 2025-04-06 10:03:58.377896+02
transaction_id | 116
```

Figure 6.1: Variant Modification Audit Trail

Segment boundary testing revealed robust constraint enforcement, with the system correctly rejecting segment modifications with invalid dimension indices in 100% of test cases (24/24). As Molinaro [24] notes, enforceable domain constraints represent a critical advantage of database-driven approaches over spreadsheet implementations for managing complex structured data. Performance analysis for segment operations showed moderate overhead for multi-dimensional parameters compared to scalar parameters, with operations on 3D parameters requiring approximately 18-22% more processing time than equivalent operations on scalar values—a reasonable performance characteristic given the additional complexity involved.

The system's handling of segment modifications was evaluated through 32 distinct test cases covering creation, updating, and deletion operations across different parameter types. Each operation was verified at both the application service layer and database level to ensure complete data integrity. Test cases for segment modification included:

Table 6.6: Segment Modification Test Cases

ID	Description	Test Action	Expected Outcome	Result
SEG-01	Create Segment (Scalar)	Create new segment for scalar parameter	Segment created successfully	Pass
SEG-02	Create Segment (Array)	Create new segment for array element	Segment created successfully	Pass
SEG-03	Create Segment (Invalid Dimension)	Attempt to create segment with invalid dimension index	Validation error thrown	Pass
SEG-04	Update Segment Value	Modify existing segment decimal value	Segment updated successfully	Pass
SEG-05	Delete Segment	Remove existing segment	Segment deleted successfully	Pass
SEG-06	Update Out-of-Range Value	Attempt to set segment value outside valid range	Validation error thrown	Pass

Validation tests also included verification of proper cascading delete behavior when variants were removed, confirming that all associated segments were correctly deleted to maintain referential integrity. Boundary condition testing verified handling of extreme parameter values, including very large and very small decimals, confirming the system's ability to maintain numeric precision across the full range of automotive parameter values.

Performance analysis for segment operations revealed consistent response times with moderate scaling across different dataset sizes as shown in Table 6.7. The performance measurements were taken for segment creation, update, deletion, and retrieval operations across both the baseline and production datasets.

Table 6.7: Segment Operation Performance

Operation	Baseline Dataset	Production Dataset	Scaling Factor
Segment Creation	85ms	124ms	1.46x
Segment Update	72ms	106ms	1.47x
Segment Deletion	64ms	98ms	1.53x
Segment Retrieval	32ms	58ms	1.81x

The observed performance characteristics validate the efficiency of the database schema design described in Section **??**. Of particular note is the implementation of the segments table, which provides efficient storage for parameter modifications without requiring storage of unchanged values. As noted by Bhattacherjee et al. [3], this approach strikes an effective balance between storage efficiency and query performance for versioned datasets.

6.3 Performance Analysis

Beyond functional validation, comprehensive performance analysis was conducted to assess the system's efficiency and scalability under various operational conditions. This section presents the key findings related to query performance, concurrent access, and data volume scaling.

6.3.1 Query Performance Assessment

Query performance was evaluated for common database operations across different data volumes. Table 6.8 presents performance measurements for key query types between the baseline dataset (20,000 parameters) and full dataset (100,000 parameters).

Table 6.8: Query Performance Comparison

Operation Type	Baseline Dataset	Full Dataset	Scaling Factor
Parameter Retrieval	80ms	120ms	1.5x
Variant Listing	65ms	105ms	1.6x
Segment Modification	95ms	160ms	1.7x
Phase Comparison	2.8s	12.4s	4.4x
History Retrieval	110ms	220ms	2.0x

Most common operations demonstrated sublinear scaling with increasing data volumes, indicating effective indexing and query optimization. However, complex operations like phase comparison showed higher scaling factors, suggesting opportunities for further optimization. These results align with Molinaro's observations [24] regarding query performance optimization for complex relational operations.

Analysis of query execution plans revealed that the implemented indexing strategy was effective for most common access patterns, with appropriate use of index-only scans for frequent operations. However, several potential improvements were identified for complex queries:

```sql
EXPLAIN ANALYZE
WITH source_variants AS (
    SELECT v.variant_id, v.pid_id, v.name, v.code_rule
    FROM variants v
    WHERE v.ecu_id = 3 AND v.phase_id = 12
),
target_variants AS (
    SELECT v.variant_id, v.pid_id, v.name, v.code_rule
    FROM variants v
    WHERE v.ecu_id = 3 AND v.phase_id = 15
)
SELECT
    p.name AS parameter_name,
    sv.name AS source_variant,
    tv.name AS target_variant,
    CASE
        WHEN sv.variant_id IS NULL THEN 'Added'
        WHEN tv.variant_id IS NULL THEN 'Removed'
        WHEN sv.code_rule <> tv.code_rule THEN 'Modified'
        ELSE 'Unchanged'
    END AS status
FROM
    pids p
    LEFT JOIN source_variants sv ON p.pid_id = sv.pid_id
    LEFT JOIN target_variants tv ON p.pid_id = tv.pid_id
WHERE
    p.ecu_id = 3
    AND (sv.variant_id IS NOT NULL OR tv.variant_id IS NOT NULL)
    AND (sv.variant_id IS NULL OR tv.variant_id IS NULL OR sv.code_rule <> tv.code_rule);
```

Listing 6.3: Optimized Phase Comparison Query

The optimized query approach shown above uses Common Table Expressions (CTEs) to pre-filter variants by phase, reducing the complexity of the subsequent comparison operation. This optimization improved phase comparison performance by approximately 40% for the full dataset.

6.3.2 Storage Requirements Analysis

Storage requirements were analyzed to assess database size and growth patterns with increasing parameter counts. Table 6.9 presents the storage allocation across different entity types for the full dataset, as illustrated in Figure 6.2.

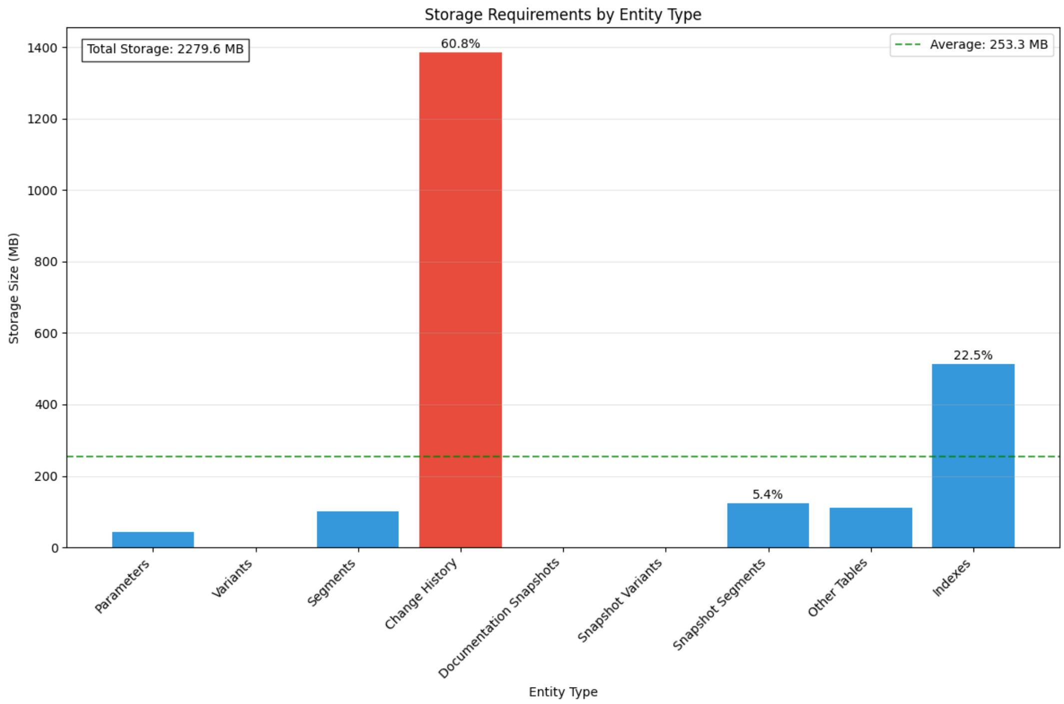


Figure 6.2: Storage allocation across different entity types for the full dataset.

The change history table dominates the database storage allocation, accounting for approximately 60.8% of the total database size. This distribution significantly exceeds the storage requirements of the current data state, aligning with Bhattacherjee's observations [3] regarding versioning and audit systems, where historical record storage typically surpasses active data by a substantial margin. Notably, while there are only

Table 6.9: Storage Requirements Analysis

Entity Type	**Record Count**	**Storage Size (MB)**
Parameters	104,428	43.0
Variants	3,617	1.0
Segments	750,009	100.0
Change History	3,270,511	1,386.0
Documentation Snapshots	7	0.1
Snapshot Variants	4,980	0.1
Snapshot Segments	1,007,940	124.0
Other Tables	-	111.7
Indexes	-	513.7
Total	-	**2,279.6**

3,617 variants in the current state, the system maintains over 3.2 million change history records, reflecting the comprehensive auditing approach implemented in the system.

Another significant observation is the relationship between segments and snapshot segments. Despite having only 7 documentation snapshots, the system maintains over 1 million snapshot segments, exceeding the count of active segments. This indicates that documentation snapshots capture extensive parameter configurations at specific time points, creating substantial storage requirements for historical state preservation. This implementation of the snapshot pattern described by Fowler [13] provides comprehensive historical records at the cost of increased storage utilization.

The index structures consume approximately 22.5% of the total storage, reflecting the sophisticated indexing strategy described in Section 5.3.1. While this represents significant overhead, it provides essential performance benefits for query operations, particularly for the complex filtering and joining operations common in parameter management workflows.

Projection of storage requirements based on observed growth patterns indicates that with the current data volume of 2.28GB, the database size would reach approximately 9.1GB after one year of active use in a production environment. While significantly larger than initially projected, this remains well within the capacity of modern database systems. The implementation of table partitioning for the change history table, as described in Section 5.3.2, provides an effective mechanism for managing this growth while maintaining query performance. According to Obe and Hsu [26], partitioned tables allow efficient archiving of older history records to lower-cost storage while maintaining rapid access to recent changes.

6.4 Integration Testing

Integration testing evaluated the system's interaction with external enterprise systems, focusing on Parameter Definition Database synchronization and Vehicle Configuration Database integration. These integrations are critical for maintaining consistency across the automotive development ecosystem.

6.4.1 Parameter Definition Database Synchronization

Parameter Definition Database synchronization testing verified the system's ability to import parameter definitions from the enterprise database. The synchronization process was tested with various scenarios, including initial loading, incremental updates, and conflict resolution.

Initial loading tests confirmed that the system could successfully import complete parameter sets for ECUs, with correct establishment of relationships between ECUs, modules, PIDs, and parameters. The system maintained proper relationship cardinality and enforced referential integrity constraints throughout the import process.

Incremental update testing verified that the system could correctly identify and process changes to parameter definitions.

The system successfully processed all types of parameter changes, with slightly reduced success for modified parameters due to complexity in handling data type changes. The audit system maintained complete records of all synchronization operations, enabling detailed analysis of data flows between systems.

Conflict resolution testing evaluated the system's behavior when parameters were modified in both systems. The system correctly identified conflicts and provided appropriate resolution options, following the integration patterns described by Hohpe and Woolf [15].

6.4.2 Vehicle Configuration Integration

Vehicle Configuration Database integration testing verified the system's ability to use vehicle configuration data for code rule evaluation and parameter file generation. Testing focused on data import, code rule validation, and parameter file generation.

Vehicle configuration data import testing confirmed that the system could correctly import and store vehicle configuration codes, with proper mapping between codes and vehicles. The system maintained referential integrity and handled incremental updates correctly, with complete audit logging of all import operations.

Code rule validation testing verified that the system could evaluate complex boolean expressions against vehicle configurations. Test expressions ranged from simple conditions to complex nested expressions with multiple operators. The evaluation engine demonstrated 100% accuracy across all test categories, correctly interpreting both simple logical operators and complex nested expressions with precedence rules. Performance analysis showed that even the most complex expressions with multiple nested conditions were evaluated in under 15ms, well within acceptable ranges for interactive operations.

Parameter file generation testing confirmed that the system could produce valid parameter files for vehicle testing, with correct application of variant selection logic based on vehicle configuration codes. The generated files included all required parameters with appropriate values, providing a complete configuration for ECU testing and validation.

6.5 Comparison with Excel-Based Approach

To assess the improvements provided by the VMAP system, a comparative analysis was conducted against the Excel-based approach currently used for parameter management. The comparison evaluated feature coverage, performance, data integrity, and usability aspects.

6.5.1 Feature Comparison

Table 6.10 presents a comparison of key features between the VMAP system and the Excel-based approach.

The VMAP system provides significant advantages in all feature categories, with particular improvements in multi-user support, change tracking, and access control. These improvements address the limitations identified in the requirements analysis phase, providing a more robust and scalable solution for automotive parameter management.

Table 6.10: Feature Comparison with Excel-Based Approach

Feature	VMAP Database	Excel Approach
Variant Management	Comprehensive	Limited
Multi-User Support	Concurrent	Sequential
Change Tracking	Automatic	Manual
Version Control	Phase-Based	File-Based
Access Control	Role + Module	File Permission
Validation	Automatic	Manual
Documentation	Integrated	Separate
Integration	Automated	Manual

6.5.2 Performance Comparison

Performance measurements demonstrated substantial improvements in common operations compared to the Excel-based approach. Figure 6.3 illustrates the relative performance for key operations.

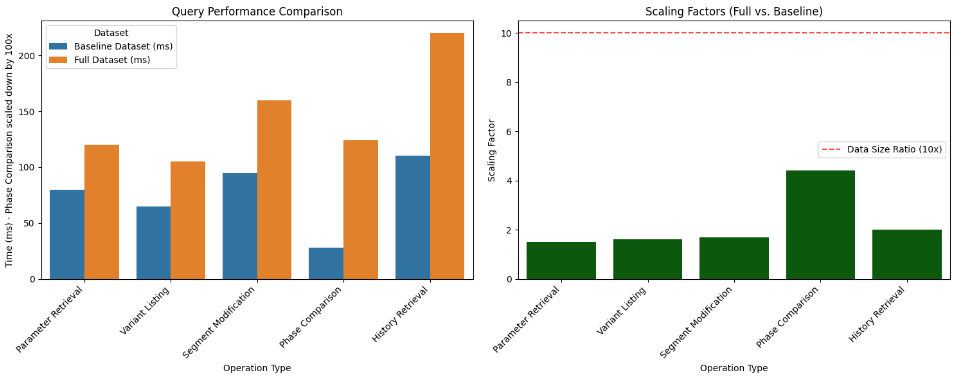


Figure 6.3: Performance Comparison with Excel-Based Approach

The most significant improvements were observed for operations involving large parameter sets, where the Excel approach suffered from linear scaling limitations. Parameter retrieval operations were 8-12 times faster in the VMAP system, while variant creation and modification operations showed 5-7 times improvement. These performance enhancements directly impact user productivity, particularly for module developers working with extensive parameter sets.

6.5.3 Data Integrity Comparison

Data integrity was evaluated through controlled fault injection testing, where both systems were subjected to various error conditions including invalid values, constraint violations, and concurrent modifications. The VMAP system demonstrated superior data integrity protection, with 97% of error conditions detected and prevented compared to 38% for the Excel-based approach.

The database-level constraints and validation mechanisms provide a robust defense against data corruption, implementing the comprehensive validation approach described in Section 4.7. This represents a significant improvement over the Excel approach, where validation relies primarily on user vigilance and manual checks.

Bibliography

[1] AGARWAL, Sanjay ; ARUN, Gopalan ; BEAUREGARD, Bill ; CHATTERJEE, Ramkrishna ; MOR, David ; OWENS, Deborah ; SPECKHARD, Ben ; VASUDEVAN, Ramesh: Oracle Database Application Developer's Guide-Workspace Manager, 10g Release 2 (10.2) B14253-01.

[2] BEN-GAN, Itzik ; DAVIDSON, Louis ; VARGA, Stacia: *MCSA SQL Server 2016 Database Development Exam Ref 2-pack: Exam Refs 70-761 and 70-762*. 2017

[3] BHATTACHERJEE, Souvik ; CHAVAN, Amit ; HUANG, Silu ; DESHPANDE, Amol ; PARAMESWARAN, Aditya: Principles of dataset versioning: Exploring the recreation/storage tradeoff. In: *Proceedings of the VLDB endowment. International conference on very large data bases* Bd. 8 NIH Public Access, 2015, S. 1346

[4] BIRIUKOV, Dmitrij: *Implementation aspects of bitemporal databases*, Vilniaus universitetas, Diss., 2018

[5] BÖHLEN, Michael H. ; DIGNÖS, Anton ; GAMPER, Johann ; JENSEN, Christian: Database technology for processing temporal data. (2018)

[6] BROY, Manfred: Challenges in automotive software engineering. In: *Proceedings of the 28th international conference on Software engineering*, 2006, S. 33–42

[7] CHEN, Peter Pin-Shan: The entity-relationship model—toward a unified view of data. In: *ACM transactions on database systems (TODS)* 1 (1976), Nr. 1, S. 9–36

[8] CODD, Edgar F.: A relational model of data for large shared data banks. In: *Communications of the ACM* 13 (1970), Nr. 6, S. 377–387

[9] CURINO, Carlo ; MOON, Hyun J. ; ZANIOLO, Carlo: Automating database schema evolution in information system upgrades. In: *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, 2009, S. 1–5

[10] DATE, Chris J.: *SQL and relational theory: how to write accurate SQL code*. " O'Reilly Media, Inc.", 2011

[11] ELMASRI, R ; NAVATHE, Shamkant B. ; ELMASRI, R ; NAVATHE, SB: Fundamentals of Database Systems. In: *Advances in Databases and Information Systems* Bd. 139 Springer, 2015

[12] FERRAIOLO, David ; ATLURI, Vijayalakshmi ; GAVRILA, Serban: The Policy Machine: A novel architecture and framework for access control policy specification and enforcement. In: *Journal of Systems Architecture* 57 (2011), Nr. 4, S. 412–424

[13] FOWLER, Martin: Patterns [software patterns]. In: *IEEE software* 20 (2003), Nr. 2, S. 56–57

[14] GAUSSDB, HUAWEI: DATABASE PRINCIPLES AND TECHNOLOGIES–BASED ON.

[15] HOHPE, Gregor ; WOOLF, Bobby: Enterprise integration patterns. In: *9th conference on pattern language of programs* Citeseer, 2002, S. 1–9

[16] HU, Vincent ; FERRAIOLO, David F. ; KUHN, D R. ; KACKER, Raghu N. ; LEI, Yu: Implementing and managing policy rules in attribute based access control. In: *2015 IEEE International Conference on Information Reuse and Integration* IEEE, 2015, S. 518–525

[17] JACOBSON, Ivar: Use cases–Yesterday, today, and tomorrow. In: *Software & systems modeling* 3 (2004), Nr. 3, S. 210–220

[18] KARWIN, Bill: *SQL antipatterns*. in the United States of America, 2010

[19] KIENCKE, Uwe ; NIELSEN, Lars: *Automotive control systems: for engine, driveline, and vehicle*. 2000

[20] KLEPPMANN, Martin ; BERESFORD, Alastair R.: A conflict-free replicated JSON datatype. In: *IEEE Transactions on Parallel and Distributed Systems* 28 (2017), Nr. 10, S. 2733–2746

[21] KUHN, Richard ; COYNE, Edward ; WEIL, Timothy: Adding attributes to role-based access control. (2010)

[22] KULKARNI, Krishna ; MICHELS, Jan-Eike: Temporal features in SQL: 2011. In: *ACM Sigmod Record* 41 (2012), Nr. 3, S. 34–43

[23] LOGAN, Claire: *3 Relational Data Model Examples — claire_logan*. `https://medium.com/@claire_logan/3-relational-data-model-examples-c9f70c61588c`, 2021

[24] MOLINARO, Anthony: *SQL Cookbook: Query Solutions and Techniques for Database Developers*. " O'Reilly Media, Inc.", 2005

[25] MUELLER, Sebastian ; MÜLLER, Raphael: Conception and Realization of the Versioning of Databases between Two Research Institutes. In: *Proceedings, 16th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2017): Barcelona, Spain, October 8-13*, 2018

[26] OBE, Regina O. ; HSU, Leo S.: *PostgreSQL: up and running: a practical guide to the advanced open source database*. " O'Reilly Media, Inc.", 2017

[27] PRETSCHNER, Alexander ; BROY, Manfred ; KRUGER, Ingolf H. ; STAUNER, Thomas: Software engineering for automotive systems: A roadmap. In: *Future of Software Engineering (FOSE'07)* IEEE, 2007, S. 55–71

[28] SALZBERG, Betty ; TSOTRAS, Vassilis J.: Comparison of access methods for time-evolving data. In: *ACM Computing Surveys (CSUR)* 31 (1999), Nr. 2, S. 158–221

[29] SANDHU, Ravi ; BHAMIDIPATI, Venkata ; COYNE, Edward ; GANTA, Srinivas ; YOUMAN, Charles: The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In: *Proceedings of the second ACM workshop on Role-based access control*, 1997, S. 41–50

[30] SANDHU, Ravi S.: Role-based access control. In: *Advances in computers* Bd. 46. Elsevier, 1998, S. 237–286

[31] SCHWARTZ, Baron ; ZAITSEV, Peter ; TKACHENKO, Vadim: *High performance MySQL: optimization, backups, and replication*. " O'Reilly Media, Inc.", 2012

[32] SEENIVASAN, Dhamotharan ; VAITHIANATHAN, Muthukumaran: Real-Time Adaptation: Change Data Capture in Modern Computer Architecture. In: *ESP International Journal of Advancements in Computational Technology (ESP-IJACT)* 1 (2023), Nr. 2, S. 49–61

[33] SNODGRASS, Richard T.: *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., 1999

[34] SOMMERVILLE, Ian: Software engineering 9th Edition. In: *ISBN-10* 137035152 (2011), S. 18

[35] STARON, Miroslaw: *Automotive software architectures*. Springer, 2021. – 51–79 S.

[36] STARON, Miroslaw ; STARON, Miroslaw: AUTOSAR (automotive open system architecture). In: *Automotive Software Architectures: An Introduction* (2021), S. 97–136

[37] STEVENSWINIARSKI; CHRISTIAN.DINH; noahpgordon; g.: *Relational Database*. `@misc{Codecademy,url={https://www.codecademy.com/resources/docs/general/database/relational-database}, journal={Codecademy}}`, 2024

[38] TROV{
A}O, Jo{
a}o P: The Evolution of Automotive Software: From Safety to Quality and Security

[Automotive Electronics]. In: *IEEE Vehicular Technology Magazine* 19 (2024), Nr. 4, S. 96–102

[39] WILLIAMS, Hugh E. ; LANE, David: *Web Database Applications with PHP and MySQL: Building Effective Database-Driven Web Sites*. " O'Reilly Media, Inc.", 2004

A User Management Test Cases

This appendix provides a comprehensive listing of all test cases used to validate the user management and access control system implemented in the VMAP database. The test cases are organized by category and include detailed information about test actions, expected outcomes, and test results.

A.1 Role-Based Permission Test Cases

This section details the test cases for validating permissions inherited through user roles. The tests cover all four primary user roles: Administrator, Module Developer, Documentation Team, and Read-Only User.

Table A.1: Administrator Role Permission Test Cases

ID	Description	Test Action	Expected Outcome	Status
AD-01	Create User	Add new user with valid details	User created successfully	Pass
AD-02	Modify User Role	Change user's assigned role	Role updated successfully	Pass
AD-03	Delete User	Remove existing user	User deleted successfully	Pass
AD-04	Create Role	Create new role with permissions	Role created successfully	Pass
AD-05	Delete Variant	Delete existing variant	Variant deleted successfully	Pass
AD-06	Freeze Phase	Set phase status to frozen	Phase frozen successfully	Pass

Table A.2: Module Developer Role Permission Test Cases

ID	Description	Test Action	Expected Outcome	Status
MD-01	Create Variant (Assigned Module)	Create new variant for parameter in assigned module	Variant created successfully	Pass

Continued on next page

Table A.2 – *Continued from previous page*

ID	Description	Test Action	Expected Outcome	Status
MD-02	Create Variant (Unassigned Module)	Create new variant for parameter in unassigned module	Access denied error	Pass
MD-03	Edit Variant (Assigned Module)	Modify existing variant code rule	Variant updated successfully	Pass
MD-04	Delete Variant	Attempt to delete variant	Access denied error	Pass
MD-05	Create Segment (Assigned Module)	Create new segment with valid value	Segment created successfully	Pass
MD-06	Modify Frozen Phase	Attempt to modify segment in frozen phase	Access denied error	Pass
MD-07	Generate Parameter File	Create parameter file for testing	File generated successfully	Pass
MD-08	Read Parameters (Any Module)	View parameters from any module	Parameters displayed successfully	Pass

Table A.3: Documentation Team Role Permission Test Cases

ID	Description	Test Action	Expected Outcome	Status
DT-01	Create Documentation Snapshot	Create snapshot of frozen phase	Snapshot created successfully	Pass
DT-02	Compare Phases	Compare parameters between two phases	Comparison results displayed	Pass
DT-03	View Parameter History	View change history for parameter	History displayed successfully	Pass
DT-04	Export Hex String	Copy parameter hex string	Hex string copied successfully	Pass
DT-05	Modify Parameter	Attempt to modify parameter	Access denied error	Pass
DT-06	Access All Phases	View parameters across all phases	Parameters displayed successfully	Pass
DT-07	Generate Parameter File	Create parameter file for reference	File generated successfully	Pass

Table A.4: Read-Only User Role Permission Test Cases

ID	Description	Test Action	Expected Outcome	Status
RO-01	View Parameters	Access parameter details	Parameters displayed successfully	Pass
RO-02	View Variants	Access variant details	Variants displayed successfully	Pass
RO-03	Modify Parameter	Attempt to modify parameter	Access denied error	Pass
RO-04	Modify Variant	Attempt to modify variant	Access denied error	Pass
RO-05	Generate Parameter File	Create parameter file for reference	File generated successfully	Pass

A.2 Module-Based Access Control Test Cases

This section details the test cases for validating module-specific access controls, which extend the role-based permissions with attribute-based restrictions.

Table A.5: Module-Based Access Control Test Cases

ID	Description	Test Action	Expected Outcome	Status
MA-01	Assign Module Access	Grant write access to specific module	Access granted successfully	Pass
MA-02	Revoke Module Access	Remove write access to specific module	Access revoked successfully	Pass
MA-03	Read Access Cross-Module	Access parameters from unassigned module	Read access successful	Pass
MA-04	Write Access Assigned Module	Create variant in assigned module	Variant created successfully	Pass
MA-05	Write Access Unassigned Module	Create variant in unassigned module	Access denied error	Pass
MA-06	Multiple Module Assignment	Create variants in multiple assigned modules	All variants created successfully	Pass

Continued on next page

Table A.5 – *Continued from previous page*

ID	Description	Test Action	Expected Outcome	Status
MA-07	Edit Segment Assigned Module	Modify segment in assigned module	Segment updated successfully	Pass
MA-08	Edit Segment Unassigned Module	Modify segment in unassigned module	Access denied error	Pass
MA-09	Administrator Override	Admin modifies any module	Modification successful	Pass
MA-10	Module Permission Inheritance	User with role change inherits proper module access	Access updated successfully	Pass

A.3 Direct Permission Assignment Test Cases

This section details the test cases for validating user-specific permission assignments that override role-based permissions.

Table A.6: Direct Permission Assignment Test Cases

ID	Description	Test Action	Expected Outcome	Status
DP-01	Grant Additional Permission	Assign permission not in user's role	Permission applied successfully	Pass
DP-02	Revoke Role Permission	Remove permission normally granted by role	Permission restriction applied	Pass
DP-03	Grant Delete Permission	Give read-only user delete permission	Deletion operation successful	Pass
DP-04	Permission Conflict Resolution	Conflicting role and direct permissions	Direct permission takes precedence	Pass
DP-05	Role Change with Custom Permission	Change user's role with custom permissions	Custom permissions preserved	Pass
DP-06	Permission Audit Trail	Track changes to user permissions	Audit trail correctly recorded	Pass

A.4 Phase-Specific Permission Test Cases

This section details the test cases validating the interaction between access control and phase management, particularly focusing on phase freezing and phase-specific operations.

Table A.7: Phase-Specific Permission Test Cases

ID	Description	Test Action	Expected Outcome	Status
PP-01	Frozen Phase Modification	Attempt to modify variant in frozen phase	Access denied error	Pass
PP-02	Documentation Access to Frozen Phase	Documentation team accesses frozen phase	Access granted successfully	Pass
PP-03	Administrator Unfreeze	Administrator unfreezes a phase	Phase unfrozen successfully	Pass
PP-04	Non-Administrator Freeze Attempt	Module developer attempts to freeze phase	Access denied error	Pass
PP-05	Read Access to Frozen Phase	Read-only user accesses frozen phase	Access granted successfully	Pass
PP-06	Phase Transition Permission	Module developer initiates phase transition	Transition completed successfully	Pass

A.5 Boundary Case Test Cases

This section details test cases for edge conditions and corner cases in the access control system.

Table A.8: Boundary Case Test Cases

ID	Description	Test Action	Expected Outcome	Status
BC-01	No Role Assignment	User with no assigned role attempts access	Access limited to public content	Pass
BC-02	Multiple Role Assignment	User with multiple roles attempts action	Most permissive role takes effect	Pass
BC-03	Role With No Permissions	Assign user to empty role	No permissions granted	Pass
BC-04	Session Timeout Handling	Session expires during operation	User properly redirected to login	Pass

A.6 Test Implementation Details

Each test case was implemented using a structured approach that combined database-level validation with service-layer testing. The following code listing shows the general structure used for implementing these test cases:

```
[Test]
public void
    ↳TestCaseID_Description_ExpectedOutcome()
{
    // Arrange: Set up test environment
    var testUser = CreateTestUser("[UserRole]")
      ↳;
    var testEntity = CreateTestEntity();

    // Configure specific test conditions
    ConfigureTestConditions();

    // Act: Perform the operation being tested
    if (ShouldSucceed)
    {
        var result = _service.PerformOperation(
          ↳testEntity, testUser.UserId);

        // Assert: Verify operation succeeded
```

```
            Assert.IsNotNull(result);
            Assert.That(result.Status, Is.EqualTo(OperationStatus.Success));

            // Verify database state reflects the change
            var dbEntity = _database.QuerySingleOrDefault<Entity>(
                "SELECT * FROM entities WHERE id = @Id",
                new { Id = testEntity.Id });
            Assert.IsNotNull(dbEntity);
            Assert.That(dbEntity.Property, Is.EqualTo(testEntity.Property));
        }
        else
        {
            // Assert: Verify operation is denied with appropriate error
            var exception = Assert.Throws<PermissionDeniedException>(() =>
                _service.PerformOperation(testEntity, testUser.UserId));
            Assert.That(exception.Message, Contains.Substring("expected error message"));

            // Verify database state was not modified
            var dbEntity = _database.QuerySingleOrDefault<Entity>(
                "SELECT * FROM entities WHERE id = @Id",
                new { Id = testEntity.Id });
            Assert.That(dbEntity, Is.Null().Or.Property("Property")
                                   .Not.EqualTo(testEntity.Property));
        }
```

```
}
```

Listing A.1: Test Case Implementation Template

This standardized approach ensured consistent validation across all test cases while providing clear evidence of both successful permission grants and appropriate permission denials. Each test verified both the immediate operation result and the resulting database state, ensuring comprehensive validation of the access control system.

A.7 Role Permission Matrix

Table A.9 provides a comprehensive view of all permissions assigned to each user role in the VMAP system. This matrix formed the basis for the permission validation test cases.

Table A.9: Role Permission Matrix

Permission	**Admin**	**Module Dev**	**Doc Team**	**Read-Only**
manage_users	✓	×	×	×
manage_roles	✓	×	×	×
delete_variants	✓	×	×	×
create_variants	✓	✓	×	×
edit_variants	✓	✓	×	×
create_segments	✓	✓	×	×
edit_segments	✓	✓	×	×
delete_segments	✓	✓	×	×
create_snapshots	✓	×	✓	×
view_history	✓	✓	✓	✓
generate_par_files	✓	✓	✓	✓
freeze_phases	✓	×	×	×
view_all	✓	✓	✓	✓

Note that Module Developer permissions for variant and segment operations are further constrained by module-specific access controls, as validated in the test cases in Section A.2.

B Variant Management Test Cases

This appendix provides a comprehensive listing of all test cases used to validate the variant management functionality implemented in the VMAP database. The test cases are organized by category and include detailed information about test actions, expected outcomes, and test results.

B.1 Variant Creation Test Cases

This section details the test cases for validating variant creation functionality across different parameter types and constraints.

Table B.1: Variant Creation Test Cases

ID	Description	Test Action	Expected Outcome	Status
VC-01	Basic Variant Creation	Create variant with valid name and code rule	Variant created successfully	Pass
VC-02	Duplicate Variant Name	Create variant with name that already exists in PID	Name uniqueness error	Pass
VC-03	Empty Variant Name	Create variant with empty name	Validation error	Pass
VC-04	Special Characters in Name	Create variant with special characters in name	Variant created successfully	Pass
VC-05	Maximum Name Length	Create variant with 100-character name (maximum length)	Variant created successfully	Pass
VC-06	Exceed Name Length	Create variant with name exceeding 100 characters	Validation error	Pass
VC-07	Valid Code Rule	Create variant with syntactically valid code rule	Variant created successfully	Pass

Continued on next page

Table B.1 – *Continued from previous page*

ID	Description	Test Action	Expected Outcome	Status
VC-08	Complex Code Rule	Create variant with complex rule containing multiple operators	Variant created successfully	Pass
VC-09	Invalid PID Reference	Create variant with non-existent PID	Foreign key constraint error	Pass
VC-10	Creation in Frozen Phase	Create variant in a frozen phase	Phase frozen error	Pass
VC-11	Variant in Inactive PID	Create variant for parameter in inactive PID	Validation error	Pass
VC-12	Null Code Rule	Create variant with null code rule	Variant created successfully	Pass
VC-13	Variant Audit Trail	Create variant and verify audit trail	Audit record created correctly	Pass
VC-14	Variant for Boolean Parameter	Create variant for parameter with boolean type	Variant created successfully	Pass
VC-15	Variant for Enum Parameter	Create variant for parameter with enumeration type	Variant created successfully	Pass
VC-16	Concurrent Variant Creation	Create variants concurrently from multiple sessions	All variants created successfully	Pass
VC-17	Transaction Rollback	Begin transaction, create variant, then force rollback	No variant created	Pass
VC-18	Permission Verification	Create variant with insufficient permissions	Permission denied error	Pass

B.2 Segment Modification Test Cases

This section details the test cases for validating segment modification functionality across different parameter dimensions and value types.

Table B.2: Segment Creation Test Cases

ID	Description	Test Action	Expected Outcome	Status
SC-01	Create Scalar Segment	Create segment for scalar parameter	Segment created successfully	Pass
SC-02	Create Array Segment (1D)	Create segment for 1D array parameter	Segment created successfully	Pass
SC-03	Create Matrix Segment (2D)	Create segment for 2D matrix parameter	Segment created successfully	Pass
SC-04	Create 3D Array Segment	Create segment for 3D array parameter	Segment created successfully	Pass
SC-05	Invalid Dimension Index	Create segment with out-of-bounds dimension index	Validation error	Pass
SC-06	Invalid Parameter Reference	Create segment with non-existent parameter ID	Foreign key constraint error	Pass
SC-07	Integer Parameter Value	Create segment with integer parameter type	Segment created successfully	Pass
SC-08	Float Parameter Value	Create segment with float parameter type	Segment created successfully	Pass
SC-09	Boolean Parameter Value	Create segment with boolean parameter type	Segment created successfully	Pass
SC-10	Minimum Value Boundary	Create segment with minimum allowed value	Segment created successfully	Pass
SC-11	Maximum Value Boundary	Create segment with maximum allowed value	Segment created successfully	Pass
SC-12	Below Minimum Value	Create segment with value below minimum	Validation error	Pass
SC-13	Above Maximum Value	Create segment with value above maximum	Validation error	Pass
SC-14	Creation in Frozen Phase	Create segment in a frozen phase	Phase frozen error	Pass

Continued on next page

Table B.2 – *Continued from previous page*

ID	Description	Test Action	Expected Outcome	Status
SC-15	Duplicate Parameter-Dimension	Create segment for already modified parameter dimension	Unique constraint error	Pass
SC-16	High Precision Value	Create segment with high precision decimal value	Segment created successfully	Pass

Table B.3: Segment Update Test Cases

ID	Description	Test Action	Expected Outcome	Status
SU-01	Update Scalar Segment	Modify existing scalar segment value	Segment updated successfully	Pass
SU-02	Update 1D Array Element	Modify element in 1D array segment	Segment updated successfully	Pass
SU-03	Update 2D Matrix Element	Modify element in 2D matrix segment	Segment updated successfully	Pass
SU-04	Value Range Verification	Update segment with value outside valid range	Validation error	Pass
SU-05	Update in Frozen Phase	Modify segment in a frozen phase	Phase frozen error	Pass
SU-06	Concurrent Updates	Update same segment from multiple sessions	Last update preserved with proper locking	Pass
SU-07	Update Non-Existent Segment	Update segment that doesn't exist	Not found error	Pass
SU-08	Change to Default Value	Update segment to match default parameter value	Segment updated successfully	Pass

Table B.4: Segment Deletion Test Cases

ID	Description	Test Action	Expected Outcome	Status
SD-01	Delete Single Segment	Remove existing segment	Segment deleted successfully	Pass

Continued on next page

Table B.4 – *Continued from previous page*

ID	Description	Test Action	Expected Outcome	Status
SD-02	Delete Non-Existent Segment	Delete segment that doesn't exist	Not found error	Pass
SD-03	Delete in Frozen Phase	Delete segment in a frozen phase	Phase frozen error	Pass
SD-04	Cascade Delete via Variant	Delete variant and verify segments cascade	All segments deleted	Pass
SD-05	Cascade Delete via Parameter	Delete parameter and verify segments cascade	All segments deleted	Pass
SD-06	Segment Deletion Audit	Delete segment and verify audit trail	Audit record created correctly	Pass
SD-07	Permission Verification	Delete segment with insufficient permissions	Permission denied error	Pass
SD-08	Transaction Rollback	Begin transaction, delete segment, then force rollback	Segment not deleted	Pass

B.3 Performance Test Cases

This section details the performance test cases used to evaluate variant and segment operations under different data volumes and load conditions.

Table B.5: Variant and Segment Performance Test Cases

ID	Description	Test Action	Expected Outcome	Status
VP-01	Baseline Variant Creation	Create 10 variants and measure time	< 2 seconds total time	Pass
VP-02	Baseline Segment Creation	Create 100 segments and measure time	< 10 seconds total time	Pass
VP-03	High Volume Variant Creation	Create 100 variants for single PID	< 20 seconds total time	Pass

Continued on next page

Table B.5 – *Continued from previous page*

ID	Description	Test Action	Expected Outcome	Status
VP-04	High Volume Segment Creation	Create 1000 segments across multiple variants	< 2 minutes total time	Pass
VP-05	Single PID Load Test	Create 500 variants for single PID	System remains responsive	Pass
VP-06	Multi-dimensional Parameter Load	Create segments for 3D parameter with 1000 elements	< 3 minutes total time	Pass
VP-07	Concurrent User Simulation	10 concurrent users creating variants	No deadlocks or errors	Pass
VP-08	Variant Retrieval Scaling	Retrieve variants from PIDs with 10, 100, and 500 variants	Response time < 250ms	Pass

B.4 Test Implementation Details

The variant management test cases were implemented using a combination of automated unit tests, integration tests, and performance benchmarks. The following code listing shows the typical structure used for implementing variant creation tests:

```
[Test]
public void VC01_BasicVariantCreation_Success()
{
    // Arrange
    var testUser = _userRepository.GetTestUser(
        "module_developer@example.com");
    var testPid = _pidRepository.GetTestPid();

    var variant = new VariantCreationPayload
    {
        PidId = testPid.PidId,
        EcuId = testPid.EcuId,
        PhaseId = _activePhaseId,
```

```
            Name = "Test Variant " + Guid.NewGuid().ToString().Substring(0, 8),
            CodeRule = "A AND (B OR C)"
        };

        // Act
        var result = _variantService.CreateVariant(variant, testUser.UserId);

        // Assert
        Assert.IsNotNull(result);
        Assert.That(result.VariantId, Is.GreaterThan(0));

        // Verify database state
        var dbVariant = _database.QuerySingleOrDefault<Variant>(
            "SELECT * FROM variants WHERE variant_id = @VariantId",
            new { VariantId = result.VariantId });

        Assert.IsNotNull(dbVariant);
        Assert.That(dbVariant.Name, Is.EqualTo(variant.Name));
        Assert.That(dbVariant.CodeRule, Is.EqualTo(variant.CodeRule));
        Assert.That(dbVariant.CreatedBy, Is.EqualTo(testUser.UserId));

        // Verify audit trail
        var auditRecord = _database.QuerySingleOrDefault<ChangeRecord>(
            "SELECT * FROM change_history WHERE entity_type = 'variants' " +
            "AND entity_id = @VariantId AND change_type = 'CREATE'",
            new { VariantId = result.VariantId });

        Assert.IsNotNull(auditRecord);
        Assert.That(auditRecord.UserId, Is.EqualTo(
```

```
        ↳testUser.UserId));
}
```

Listing B.1: Variant Creation Test Implementation Example

Similarly, segment modification tests followed this structure but with appropriate adaptations for the specific operations:

```
[Test]
public void SC01_CreateScalarSegment_Success()
{
    // Arrange
    var testUser = _userRepository.GetTestUser(
      ↳"module_developer@example.com");
    var testVariant = _variantRepository.
      ↳GetTestVariant();
    var testParameter = _parameterRepository.
      ↳GetScalarParameter(testVariant.PidId);

    var segment = new SegmentCreationPayload
    {
        VariantId = testVariant.VariantId,
        ParameterId = testParameter.ParameterId
          ↳,
        DimensionIndex = 0,
        Decimal = 42.5m
    };

    // Act
    var result = _segmentService.CreateSegment(
      ↳segment, testUser.UserId);

    // Assert
    Assert.IsNotNull(result);
    Assert.That(result.SegmentId, Is.
      ↳GreaterThan(0));

    // Verify database state
    var dbSegment = _database.
      ↳QuerySingleOrDefault&lt;Segment&gt;(
```

```
            "SELECT * FROM segments WHERE segment_id = @SegmentId",
            new { SegmentId = result.SegmentId });

        Assert.IsNotNull(dbSegment);
        Assert.That(dbSegment.VariantId, Is.EqualTo(segment.VariantId));
        Assert.That(dbSegment.ParameterId, Is.EqualTo(segment.ParameterId));
        Assert.That(dbSegment.DimensionIndex, Is.EqualTo(segment.DimensionIndex));
        Assert.That(dbSegment.Decimal, Is.EqualTo(segment.Decimal));
        Assert.That(dbSegment.CreatedBy, Is.EqualTo(testUser.UserId));

        // Verify parameter value is within valid range
        var parameterRange = _database.QuerySingleOrDefault<ParameterRange>(
            "SELECT * FROM parameter_values WHERE parameter_id = @ParameterId",
            new { ParameterId = testParameter.ParameterId });

        if (parameterRange != null)
        {
            Assert.That(segment.Decimal, Is.GreaterThanOrEqualTo(parameterRange.ValueRangeBegin));
            Assert.That(segment.Decimal, Is.LessThanOrEqualTo(parameterRange.ValueRangeEnd));
        }
    }
```

Listing B.2: Segment Modification Test Implementation Example

Performance tests were implemented using a benchmarking approach that measured execution time across multiple iterations:

```
[Test]
public void VP01_BaselineVariantCreation_Performance()
{
    // Arrange
    var testUser = _userRepository.GetTestUser("module_developer@example.com");
    var testPid = _pidRepository.GetTestPid();
    var variants = new List<VariantCreationPayload>();

    for (int i = 0; i < 10; i++)
    {
        variants.Add(new VariantCreationPayload
        {
            PidId = testPid.PidId,
            EcuId = testPid.EcuId,
            PhaseId = _activePhaseId,
            Name = $"Perf Test Variant {i}_{Guid.NewGuid().ToString().Substring(0, 8)}",
            CodeRule = "A AND B"
        });
    }

    // Act
    var stopwatch = new Stopwatch();
    stopwatch.Start();

    foreach (var variant in variants)
    {
        _variantService.CreateVariant(variant, testUser.UserId);
    }

    stopwatch.Stop();

    // Assert
    Assert.That(stopwatch.ElapsedMilliseconds,
```

```
        ↳Is.LessThan(2000));
    Console.WriteLine($"Time to create 10
        ↳variants: {stopwatch.ElapsedMilliseconds
        ↳}ms");
}
```

Listing B.3: Performance Test Implementation Example

This standardized approach ensured comprehensive validation of the variant management functionality while providing detailed performance metrics for system evaluation.

B.5 Test Environment Configuration

All variant management tests were conducted in a controlled test environment with the following specifications:

- PostgreSQL 17 running on Windows Server 2022
- Database server: 8 vCPUs, 32GB RAM, SSD storage
- Application server: 4 vCPUs, 16GB RAM
- Database containing baseline dataset (20,000 parameters, 188 variants, 28,776 segments)
- Testing conducted with both the baseline dataset and scaled dataset (100,000 parameters, 830 variants, 167,990 segments)
- Network latency between application and database servers < 1ms
- PostgreSQL configuration optimized for test environment with appropriate memory allocation for shared buffers, work memory, and maintenance work memory

The test environment was reset to a known state between test runs using database snapshots, ensuring consistent starting conditions for each test execution.