Automated documentation creation based on implemented data models: An approach to improve user information, process conformance and configuration maintenance

MASTER THESIS

SANDEEPKUMAR RAMESH

Submitted on 07 January 2026

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A – FAU Erlangen
Mr. Alexander Scholz – Valeo Erlangen

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Erlangen, 07 January 2026

License

_____

Erlangen, 07 January 2026

Abstract

This thesis presents a comprehensive approach to addressing the pervasive challenges of manual documentation within complex software development, with a specific focus on the automotive sector. The core of this research involves the design and implementation of a React JS-based widget seamlessly integrated into the Polarion Application Lifecycle Management (ALM) environment. The document details the widget's architecture, which leverages Polarion robust data models and API capabilities, and its practical implementation. The anticipated benefits include significant improvements in user information accessibility, enhanced process conformance, streamlined configuration maintenance, and an overall positive impact on company performance. A central tenet of this work is the widget's adherence to stringent automotive industry standards, notably ISO 26262 and Automotive SPICE (ASPICE), demonstrating its capacity to foster efficiency and regulatory compliance simultaneously.

This work contributes to the field of software engineering by establishing a framework for integrating modern web technologies with enterprise ALM systems, providing a blueprint for similar implementations across different platforms, and quantifying the organizational benefits of automated documentation generation.

Acknowledgements

I would like to express my deepest and most sincere gratitude to my project supervisor, Alexander Scholz. His invaluable guidance and steady support were the cornerstones of this research process. His profound expertise and unwavering dedication to excellence were instrumental in shaping the direction of this project, and I am truly grateful for the intellectual rigor he encouraged in my work.

I also extend my heartfelt thanks to Dr. Dirk Riehle, my university supervisor at the Friedrich-Alexander-Universität Erlangen-Nürnberg. His academic oversight and high standards for research in Open Source Software provided the necessary framework for this thesis to meet the rigorous requirements of a Master of Science in Computer Science.

A very special note of thanks goes to my Manager from the Valeo HV Inverter TPM team, Mr. Eric Heinze. His continuous support, mentorship, and belief in the value of this research from its initial conception to its final completion were vital to my success. Balancing professional responsibilities with a Master's thesis is a significant undertaking, and his patience and insight provided me with the supportive environment I needed to thrive to complete the thesis to the fullest.

Table of contents

Index

Introduction

The automotive industry is undergoing a paradigm shift toward software-centric innovation, where digital systems define not only functionality but also safety, efficiency, and user experience. This transformation has amplified the need for precise, up-to-date, and traceable documentation throughout the product lifecycle. In safety-critical environments, documentation is more than a regulatory requirement; it is the operational backbone ensuring that every design decision, configuration change, and test result is fully recorded for auditability.[9] However, traditional manual documentation practices have reached a point of obsolescence, failing to match the speed and complexity of modern development methodologies.

However, traditional manual documentation practices have not evolved at the same pace as software development methodologies. Agile, DevOps, and continuous delivery frameworks prioritize rapid iteration, while manual documentation processes remain static, error-prone, and resource-intensive.[12] This mismatch creates a structural inefficiency: documentation becomes a bottleneck rather than an enabler of process quality. Particularly in enterprise-scale environments managed through platforms like Polarion Application Lifecycle Management (ALM)[28], maintaining synchronization between real-time development activities and formal documentation is both challenging and costly.

To address this, the present thesis explores the development of an automated documentation framework integrated directly within Polarion ALM, leveraging React JS as a modern, modular front-end technology. The objective is to transform documentation from a passive, after-the-fact deliverable into a dynamic and continuously updated artifact, directly reflecting the live state of implemented data models. By utilizing Polarion's APIs and data structures, the system generates structured, auditable documents that mirror current configurations and workflows.

The proposed solution aims to reduce redundant manual effort, ensure compliance with automotive standards such as ISO 26262 and Automotive SPICE (ASPICE), and improve cross-functional collaboration between development, testing, and quality-assurance teams. Beyond efficiency, the integration demonstrates how modern web technologies can bridge legacy enterprise systems and contemporary software-engineering practices, offering a scalable model for digital documentation transformation.

1.1 Challenges of Manual Documentation in Software Development

A primary challenge lies in the persistent mismatch between Agile delivery cycles and static documentation updates. While Agile and DevOps frameworks prioritize rapid iteration and continuous integration, manual documentation remains a resource-intensive bottleneck that disrupts these feedback loops. Without automation, documentation becomes a reactive, after-the-fact deliverable rather than a dynamic component integrated within the implementation process. This structural inefficiency slows down the entire software development lifecycle and erodes the agility that modern engineering teams strive to achieve.

The phenomenon of "documentation drift" represents one of the most severe risks in manual systems.[6] As design elements, configurations, and test artifacts are updated frequently, corresponding manual records seldom reflect the actual system state. This drift causes discrepancies between the implemented logic and recorded specifications, undermining the trust that engineers and auditors place in project materials. In automotive contexts, such misalignment can delay Functional Safety (ISO 26262) audits or ASPICE maturity assessments, as minor omissions in traceability can invalidate compliance evidence.

Beyond compliance risks, the resource intensity of manual documentation is a major hindrance to innovation. Creating and maintaining documents manually requires extensive engineering hours that could otherwise be dedicated to development or verification tasks. Each iteration typically necessitates multiple review cycles across development, validation, and quality assurance departments, leading to duplicated efforts and fragmented accountability. This is further complicated by the use of varying templates and inconsistent terminology, which complicates downstream analysis and automated parsing.

Finally, manual methods introduce significant information-management challenges regarding standardization and visibility. Textual documents stored in shared drives or local repositories often lack proper version control and are difficult for distributed teams to locate. When changes propagate through interconnected systems, identifying all affected documents becomes impractical, increasing operational risk. To maintain process integrity, there is a clear need for a transition toward automated, data-driven documentation that is tightly coupled with the tools and workflows that generate the underlying project data.

1.2 Research Motivation

The motivation for this research is rooted in the urgent need to bridge the persistent gap between documentation accuracy and the rapid evolution of complex automotive software projects. While the shift toward digital transformation is a recognized industry goal, organizations utilizing enterprise platforms like **Polarion Application Lifecycle Management (ALM)** still struggle with manual or semi-automated processes to export configuration data. These legacy practices are inherently slow and result in fragmented documentation sets that lack consistent formatting and real-time synchronization with the project's actual state.

A decisive factor driving this research is the **technical and functional inadequacy of existing legacy widgets** within the Polarion environment. Current server-rendered interfaces, often based on Java, suffer from a significant lack of UI/UX sophistication, making it difficult to effectively visualize the high density of information required for modern data models. These existing tools are frequently static and non-interactive, failing to provide stakeholders with a clear, intuitive view of interconnected workflow states and multi-layered link roles. This lack of visual clarity creates "friction" that hinders cross-functional collaboration and slows down the audit preparation process.

This research is therefore motivated by the potential of **React JS** to revolutionize this landscape by transforming documentation into a "living," dynamic artifact. By leveraging Polarion’s robust APIs and modular frontend technology, this project seeks to replace restrictive, legacy interfaces with a high-performance, component-based widget. The motivation is to establish a framework that ensures documentation is a "single source of truth," synchronized with the system's actual configuration data. Ultimately, this work aims to demonstrate how modern web technologies can bridge the gap between legacy enterprise systems and contemporary software engineering, fostering efficiency and regulatory compliance simultaneously.

2. Objectives

The overarching goal of this research is to design, develop, and evaluate an **automated documentation generation system** integrated within the **Polarion Application Lifecycle Management (ALM)** environment. The solution leverages **React JS** for front-end automation and Polarion's **REST and Java APIs** for data extraction and configuration management. By doing so, it aims to bridge the persistent gap between documentation accuracy and system evolution in complex automotive software projects.

This work aligns with three central aims:

1. To **reduce human dependency** in documentation processes by embedding automation directly into enterprise development workflows.
2. To **ensure continuous alignment** between implemented configurations, requirements, and generated documentation.
3. To **improve process efficiency and compliance** through dynamic documentation that supports automotive quality standards such as ISO 26262 and ASPICE.

The proposed framework positions documentation not as a static record but as a **continuously updated artifact** : a ―single source of truth‖ synchronized with the system's actual configuration data. This contributes to faster validation cycles, enhanced transparency, and reduced audit preparation time.

2.1 Problem Statement

Despite continuous technological progress in software development tools and methodologies, **documentation remains a weak link** in the software lifecycle, particularly in regulated domains such as the automotive industry. Current practices in organizations using Polarion ALM involve manual or semi-automated export of work items and configuration data into templates. These manual processes are slow, error-prone, and difficult to scale. The results are fragmented documentation sets that lack version synchronization, consistent formatting, and traceability.

Key issues identified include:

1. **Documentation Latency:** Manual updates lag behind frequent system changes, causing discrepancies between actual implementation and recorded artifacts.
2. **Resource Overhead:** Skilled engineers spend significant time maintaining

documentation instead of focusing on development and validation.

3. **Inconsistency Across Teams:** Variations in documentation style, template usage, and terminology weaken cross-departmental alignment.
4. **Limited Tool Integration:** Documentation tools often operate independently of ALM systems, breaking traceability links and increasing compliance risk.
5. **Verification Complexity:** Validating whether documents truly represent the current system configuration is tedious and often requires manual cross-checking.

These limitations become particularly critical in environments governed by **ISO 26262 functional safety** and **ASPICE process maturity** frameworks, where documentation completeness and traceability directly affect certification. Consequently, there is a need for a **robust, automated, and maintainable system** that generates and updates documentation dynamically based on the system's underlying data models.

This research therefore proposes a **React JS-based Polarion widget** that automates documentation generation by directly interfacing with Polarion's APIs. The solution addresses the dual challenge of **maintaining documentation accuracy** and **enhancing process efficiency**, while adhering to enterprise security and performance constraints.

2.2 Research Questions

To achieve these objectives, the study is guided by the following research questions (RQs):

- **RQ1:** How can React JS be effectively integrated within Polarion ALM to enable automated documentation generation based on implemented data models?
- **RQ2:** Which combination of Polarion's REST and Java APIs provides optimal balance between accessibility, security, and performance for real-time data extraction?
- **RQ3:** To what extent does automated documentation improve efficiency, accuracy, and user satisfaction compared to traditional manual documentation workflows?
- **RQ4:** What architectural and design patterns best support the integration of modern web technologies with enterprise-grade ALM systems?
- **RQ5:** How does the adoption of automated documentation influence organizational factors such as process conformance, configuration maintenance, and audit readiness?

Each question addresses a distinct dimension of the research from technical integration and

architectural design to operational impact and compliance benefits. Collectively, these questions structure the inquiry and evaluation methodology, ensuring that both the **engineering feasibility** and **organizational implications** of automation are rigorously examined.

3. Literature Review

Software documentation has been an enduring focus of software-engineering research for decades, evolving from simple text manuals to dynamic, model-linked representations of complex systems. The academic study of software documentation has transitioned from a focus on static manuals toward dynamic, model-linked representations. This review synthesizes key findings across twelve fundamental and contemporary sources to establish a rigorous foundation for the proposed automated system.

3.1 The Role of Documentation in Software Engineering

Documentation serves as both a primary communication medium and a vital regulatory artifact within the engineering lifecycle.

- **Parnas and Clements (1986):** These authors emphasized that accurate documentation is a cornerstone for system maintainability and design comprehension, particularly as knowledge is passed through generations of engineers.[15]
- **Forward and Lethbridge (2002):** Their research categorized documentation into four distinct domains requirements, architectural, technical, and user noting that each serves unique audiences and specific project purposes.[4]
- **Lethbridge et al. (2003):** Through empirical studies, these researchers revealed a critical trend where documentation quality often deteriorates over time, often leading developers to perceive it as outdated or incomplete.[12]
- **Visconti and Cook (2002):** This study defined the "documentation dilemma," which describes the constant trade-off engineers face between producing detailed, high-quality records and meeting tight delivery schedules.[23]
- **Garousi et al. (2015):** Their work demonstrated that the primary cost driver in documentation is maintenance rather than initial creation, identifying "documentation drift" as the result of system evolution outpacing static updates.[6]
- **Robillard et al. (2017) & Dagenais and Robillard (2014):** These scholars advocate for "just-in-time" documentation and generation from model metadata to minimize human error and ensure synchronization.[17]

3.2 Automated Documentation Systems and "Living" Artifacts

Automation represents the essential step in overcoming the manual inefficiencies inherent in traditional engineering practices.

- **Sriplakich et al. (2008):** They defined automated documentation as the algorithmic generation of descriptive artifacts directly from software structures, execution data, or underlying models.[21]
- **Schärli et al. (2015):** This research introduced the "living documentation" paradigm, wherein documentation is not a static file but a dynamic entity that evolves continuously alongside the system.[19]
- **Roehm et al. (2012):** These scholars classified automation techniques into three primary categories: code-to-documentation (e.g., Javadoc), model-to-documentation (e.g., UML), and runtime-to-documentation (e.g., execution traces).[18]

3.3 Modern Web Technologies in Enterprise Applications

The transition of enterprise systems toward hybrid and cloud architectures has fuelled the integration of modern web frameworks for internal tool development[14].

- **Fielding (2000):** In his foundational work, Fielding defined Representational State Transfer (REST) as a stateless, scalable, and cacheable communication protocol over HTTP.[3]
- **Pautasso et al. (2008):** These authors explored architectural trade-offs, concluding that the simplicity and alignment of REST with web standards make it the ideal choice for data-centric enterprise integrations.[16]
- **Graziotin and Abrahamsson (2013):** Their research highlighted how modular frameworks like Angular, Vue, and React JS significantly accelerate development and promote cross-platform compatibility.[7]
- **Maddigan et al. (2018):** This study demonstrated that React can effectively serve as the engine for dynamic documentation portals, enabling high levels of component reusability and real-time data visualization.[13]

3.4 The Shift to Declarative UI and Virtual DOM Performance

A critical advancement in web technology is the move from imperative DOM manipulation to declarative UI frameworks.

- **Fedosejev (2015) & Gackenheimer (2015):** These researchers documented the rise of React JS as a dominant force due to its virtual DOM, which allows for high-speed rendering of complex datasets by only updating changed elements.[5]
- **Daigneau (2011) & Newman (2015):** These authors advocate for the Backend-for-Frontend (BFF) pattern and microservices architectures, which allow APIs to be structured to serve the unique needs of specific frontend interfaces like the proposed Polarion widget.[14]

3.5 Extensibility Frameworks in Application Lifecycle Management (ALM)

Polarion ALM's architecture provides a unique environment for the integration of modern web technologies.

- **Kääriäinen and Välimäki (2009):** They described Polarion's repository-centric architecture as a unifying structure that ensures version control and data integrity across the entire product lifecycle.[10]
- **Kleebaum et al. (2018) & Hegedüs et al. (2016):** These studies demonstrated how Polarion's Java API and REST API can be leveraged to create model-driven engineering tools and lightweight client-side applications.[11]

3.6 Comparison of Frontend Frameworks for Enterprise Widgets

To justify the selection of React JS, it is necessary to compare it with other market-leading technologies within the context of enterprise integration[23].

Feature	**React JS**	**Angular**	**Vue.js**
Philosophy	Flexible UI Library	Opinionated MVC Framework	Progressive Framework
Bootstrapping	Simple DOM injection	Heavyweight/Rigid	Incremental Adoption

Feature	React JS	Angular	Vue.js
Performance	High (Virtual DOM)	Robust but high overhead	Lightweight and fast
Learning Curve	Moderate	Steep (RxJS/TypeScript)	Gentle

Table 3.1: Comparative Analysis of Frontend Frameworks (React JS, Angular, Vue.js). *This table contrasts the three market-leading frameworks based on philosophy, bootstrapping complexity, performance, and learning curve to justify the selection of React JS.*

Justification for React:

- **Lightweight Integration:** Unlike Angular, which is rigid and heavyweight, React's simple ReactDOM.render() bootstrapping method makes it ideal for embedding within a Polarion Rich Page rendered by Velocity.
- **Ecosystem and Maintenance:** React's larger developer ecosystem and extensive library of third-party components provide greater long-term assurances for enterprise maintainability and talent acquisition at Valeo.
- **Performance:** React's virtual DOM is particularly suited for rendering the massive datasets and complex link role graphs typical of automotive project configurations.

3.7 Research Gap

The reviewed literature highlights several unexplored areas that this research seeks to address:

- **Systematic ALM-Frontend Integration:** While React is prevalent in web apps, its deep integration into the Polarion ecosystem remains largely undocumented.
- **Process-Regulated Automation:** Existing studies focus on code-level extraction rather than enterprise-wide lifecycle systems where ASPICE and ISO 26262 compliance are mandatory.
- **Hybrid API Orchestration:** Most literature treats REST and Java APIs as independent entities; this work leverages both to balance data integrity with frontend responsiveness.
- **The Velocity-React Bridge:** Using Velocity templates as a secure, server-side data-transfer layer to bootstrap a modern React application is a novel architectural pattern introduced in this study.

4. React JS benefits

The selection of a frontend framework for an enterprise widget embedded within a host platform like Polarion is a critical architectural decision. The choice directly impacts performance, maintainability, and the developer's ability to integrate with the host's non-standard data-passing mechanisms. The evaluation centered on three market leaders: React JS, Angular, and Vue.js.

- **Angular:** As a comprehensive, "opinionated" framework, Angular (maintained by Google) provides a complete Model-View-Controller (MVC) solution out of the box. Its strengths lie in its robust, scalable architecture and its enforcement of TypeScript, which enhances code quality in large teams. However, for this project's requirements, Angular was deemed too heavyweight. Its steep learning curve and rigid structure (e.g., reliance on RxJS for state management) would add unnecessary complexity to a widget that must be lightweight, fast-loading, and able to "bootstrap" from a simple <script> tag injected by a Velocity template.
- **Vue.js:** Vue is often praised for its gentle learning curve, excellent documentation, and performance. Its "progressive framework" model, allowing it to be adopted incrementally, made it a very strong contender. It can be easily embedded in a simple script tag, much like jQuery. However, in the context of a large enterprise like Valeo, React's larger developer ecosystem, extensive library of third-party components, and strong corporate backing (by Meta) provide greater long-term assurances for maintainability and talent acquisition.
- **React JS:** React was ultimately selected as the optimal choice. Its core strength is its "library" (not a framework) approach, offering high flexibility. Its component-based model perfectly aligns with the project's need for a reusable, modular UI . Most importantly, React's small footprint and simple ReactDOM.render() bootstrap method make it ideal for embedding within another application's DOM in this case, a Polarion Rich Page rendered by Velocity. Its virtual DOM ensures high performance when rendering the large datasets (e.g., workflow transitions, link role graphs) extracted from Polarion .

In summary, while Angular is too rigid for this type of integration and Vue is a viable alternative, React provides the best balance of flexibility, performance, and enterprise-grade support required for this project.

4.1 Constraints

Every enterprise tool operates within environmental and technical boundaries. These constraints were identified early to avoid design conflicts with Polarion's architecture.

- C1: The widget must be compatible with Polarion version 3.20.2 or newer.
- C2: Supported browsers include Chrome 80+, Firefox 75+, and Microsoft Edge 80+.
- C3: The widget must not modify Polarion's database schema or interfere with core operations.
- C4: Deployment must adhere to Siemens' widget packaging guidelines.
- C5: Integration must not degrade server performance or conflict with other custom widgets.
- C6: The design must align with Valeo's cybersecurity and access-control policies.

4.2 Automotive Industry Standards: ISO 26262 and ASPICE

The automotive industry's reliance on complex software demands strict compliance with standards like **ISO 26262** and **ASPICE** to ensure safety and quality.[22] ISO 26262 mandates thorough documentation across the development lifecycle for E/E systems, including safety plans, test reports, and change management records many of which are supported through Polarion's certified templates.[9] ASPICE complements this by defining process maturity levels (0–5), requiring structured, well-documented, and traceable workflows, particularly at Levels 2 and 3.[22]

Manual documentation often inconsistent and outdated hinders progress toward these levels. In contrast, **automated documentation** ensures consistent version control, reduced errors, and standardization, directly supporting compliance and audit-readiness. This makes the proposed React-based widget a strategic enabler for improving ASPICE maturity, process efficiency, and competitive positioning in the automotive sector.

4.3 React JS in Enterprise UI/UX and Best Practices

React JS is a widely adopted framework in modern web development, especially in enterprise UI/UX design, due to its simplicity, efficiency, and modular architecture. It supports clean, maintainable, and reusable code structures, making it ideal for scalable applications like automated documentation widgets. By using well-defined project layouts and reusable components, React helps streamline development and enhance team collaboration.

React encourages a modular approach where components are built to be small, focused, and adaptable to different use cases. Best practices include organizing files logically, adhering to the "one function = one component" rule, and applying the DRY (Don’t Repeat Yourself) principle to eliminate redundancy. The transition from class-based to functional components, along with the use of React Hooks, simplifies code and state management.

Hooks like useState and useEffect play a crucial role in managing dynamic data and side effects, which is essential for documentation widgets that interact with Polarion APIs. This component-based model makes the widget extensible allowing different documentation modules (e.g., requirements, test results) to be created and updated independently.

For large datasets and performance-critical environments, React offers built-in optimizations like the Virtual DOM, batch updates, and memoization techniques. These enable responsive rendering even with complex documentation structures.[2]

- useEffect - Used to fetch data from Polarion APIs on mount or when inputs change.
- useState - Manages retrieved data and ensures UI re-renders on updates.

4.4 Hybrid Integration Platforms (HIP) and API Integration Patterns

A **Hybrid Integration Platform (HIP)** enables seamless connectivity between cloud, on-premise, and hybrid systems supporting complex integrations, real-time data sharing, and business process automation. HIPs enhance agility, scalability, and security while offering tools for workflow design and API management.

In modern development, **API integration** is essential for enabling dynamic, secure data exchange. React applications commonly use **Fetch API** or **Axios** within useEffect to interact with RESTful APIs, making them naturally suited for data-driven tools like documentation widgets.

The proposed Polarion widget functions like a HIP component by integrating internal data models to auto-generate documentation. It not only pulls live data but also contributes to broader organizational data strategies by supporting interoperability with other systems through Polarion's APIs.

Ultimately, the widget's success depends on how efficiently it retrieves and processes data. Choosing between REST and Java APIs, optimizing queries, and managing large data volumes are critical to ensuring performance and a responsive user experience. This positions the widget as a scalable and enterprise-ready integration solution.

5. Requirements Analysis

The success of any enterprise software system depends on a clear, structured definition of requirements. For the proposed **automated documentation generation widget**, requirements were elicited through a combination of stakeholder interviews, analysis of existing documentation workflows at Valeo, and examination of Polarion ALM's internal extensibility features. This chapter presents a systematic breakdown of **functional**, **non-functional**, and **technical constraints**, providing the foundation for architectural design and evaluation.

The approach follows a semi-structured methodology inspired by the IEEE 830 standard for software requirements specification, ensuring that each requirement can later be **mapped to design components and validation results**. The requirements were reviewed collaboratively with Polarion administrators and software engineers to guarantee technical feasibility and organizational alignment.

5.1 Functional Requirements

Functional requirements describe the specific system behaviors necessary to achieve automation, traceability, and compliance. They are derived from observed bottlenecks in manual documentation and from end-user needs within the ALM environment.

5.1.1 Documentation Generation

The primary goal of the widget is to automatically generate accurate and up-to-date documentation from Polarion's existing data structures. Each functionality supports traceability, efficiency, or maintainability.

FR1.1: The system shall automatically extract Polarion data models (e.g., work items, workflows, enumerations) to reflect live configurations.

FR1.2: The widget shall generate documentation layouts dynamically, ensuring consistency with corporate templates and formatting standards.

FR1.3: Extracted metadata (attributes, relationships, constraints) shall populate the generated documentation automatically.

FR1.4: The system shall visualize complex relationships, such as workflow dependencies or hierarchical links, using interactive diagrams.

FR1.5: Users shall be able to export documentation in multiple formats HTML for internal review, PDF for audit storage, and Word for editable deliverables.

FR1.6: The widget shall perform *incremental updates*, regenerating only the affected sections when model changes occur to preserve prior custom edits.

Justification:

This functionality directly addresses the most resource-intensive element of manual documentation: the repetitive re-entry of data from ALM repositories. By mapping Polarion's artifacts to structured templates, the widget creates a *single source of truth* for project documentation, drastically reducing redundancy and ensuring real-time accuracy.

5.1.2 User Interface Requirements

The user interface (UI) serves as the bridge between complex backend data and human comprehension. The system must be intuitive, responsive, and adaptive to user expertise levels.

FR2.1: Provide a dashboard summarizing document generation status, modification history, and metadata integrity.

FR2.2: Include interactive navigation and search functions to locate data by work-item type, category, or relationship.

FR2.3: Allow configurable views and layout customization to match different project requirements or user preferences.

FR2.4: Support real-time interaction such as filtering or highlighting dependencies to visualize relationships clearly.

FR2.5: Offer a feedback channel to capture user-reported issues or enhancement requests directly within the widget.

FR2.6: Ensure full responsiveness across browsers and screen resolutions.

Justification:

Adoption of new internal tools often hinges on usability rather than capability. A responsive React-based UI ensures minimal training effort, encourages adoption across

distributed teams, and demonstrates the strength of modern web technologies within a traditionally Java-heavy environment.

5.1.3 Data Access and Integration

Integration reliability is central to ALM automation. The widget must seamlessly interact with Polarion's APIs to retrieve, transform, and display data while adhering to strict access controls.

FR3.1: Retrieve work-item structures, custom fields, and configuration settings using Polarion's REST API.

FR3.2: Use the Java API for complex or restricted data not exposed via REST, such as repository configurations.

FR3.3: Implement secure, session-based authentication to inherit user privileges and prevent unauthorized access.

FR3.4: Detect data-model changes automatically and trigger documentation updates accordingly.

FR3.5: Transform raw API responses into human-readable, structured documentation using React components and Velocity templates.

FR3.6: Ensure efficient data exchange between the backend and React frontend through a lightweight data-transfer layer.

Justification:

This multi-layer data-access design achieves a critical balance between performance and data governance. Using both REST and Java APIs leverages Polarion's full capability while maintaining data integrity, scalability, and auditability.

5.2 Non-Functional Requirements

Non-functional requirements define system attributes performance, security, maintainability, and usability that determine the software's long-term viability. These were prioritized through a stakeholder ranking process emphasizing reliability, efficiency, and compliance.

5.2.1 Performance

- **NFR1.1:** The widget shall render all visual elements and documentation views within 2 seconds under normal load.
- **NFR1.2:** Full documentation generation shall complete in under 5 minutes for large projects (10,000+ work items).
- **NFR1.3:** The system shall support at least 50 concurrent users without noticeable degradation.
- **NFR1.4:** The CPU and memory utilization shall not exceed 20% of Polarion server capacity under peak load.

  Justification:

  High responsiveness is crucial for user adoption and system integration. React's virtual DOM and lazy-rendering capabilities directly support these goals, ensuring scalability across projects of varying size and complexity.

5.2.2 Security

- **NFR2.1:** The widget shall use Polarion's internal authentication and authorization model to ensure controlled data access.
- **NFR2.2:** All communication between frontend and backend must be encrypted via HTTPS.
- **NFR2.3:** Sensitive configuration data shall never be cached on the client side.
- **NFR2.4:** The solution shall comply with Valeo's internal IT security guidelines and GDPR regulations.

Justification:

Because documentation often includes proprietary and safety-critical content, the design prioritizes secure token handling and encryption, ensuring that automation enhances rather than compromises compliance.

5.2.3 Maintainability

- **NFR3.1:** The architecture shall use a modular component structure, allowing independent updates to logic, UI, and API layers.
- **NFR3.2:** Automated unit and integration tests shall cover at least 80% of source code.

- **NFR3.3:** The system shall remain compatible with future Polarion versions through standardized API usage.
- **NFR3.4:** Dependencies shall be managed using version pinning and CI/CD testing pipelines.

Justification:

Maintainability ensures sustainability beyond the scope of the thesis. Modular code using React hooks and reusable components supports easy debugging, upgrade cycles, and future feature expansion.

5.2.4 Usability

- **NFR4.1:** Experienced users shall be able to generate documentation with minimal manual input (<3 clicks).
- **NFR4.2:** The UI shall follow intuitive navigation principles consistent with Polarion's design philosophy.
- **NFR4.3:** Tooltips, inline help, and example outputs shall support user onboarding.

Justification:

Usability requirements ensure accessibility for all project participants from engineers to auditors aligning with ISO 9241-210 human-centered design standards.

5.3 Limitations

Despite robust integration, the system is subject to certain operational limitations:

- **L1:** Performance depends on Polarion's REST and Java API throughput, which may limit scalability under heavy load.
- **L2:** Browser rendering complexity may constrain the level of visualization detail achievable for very large datasets.
- **L3:** Custom scripting restrictions in Polarion may limit dynamic interaction with certain system modules.

5.4 Evaluation:

These limitations provide scope for future optimization. For instance, caching or asynchronous API batching can mitigate throughput constraints, while selective data virtualization can enhance responsiveness without overwhelming the client interface.

6 System Design and Architecture

6.1 Overview of the Proposed Widget Solution

The proposed solution is a React JS-based widget embedded within the Polarion ALM environment to dynamically generate detailed documentation and configuration reports. It utilizes Polarion's internal data models and configurations to automate and streamline documentation, improving accuracy, consistency, and compliance especially with automotive standards. The widget features an intuitive interface that allows users to select parameters and view generated documents directly within Polarion's rich pages.

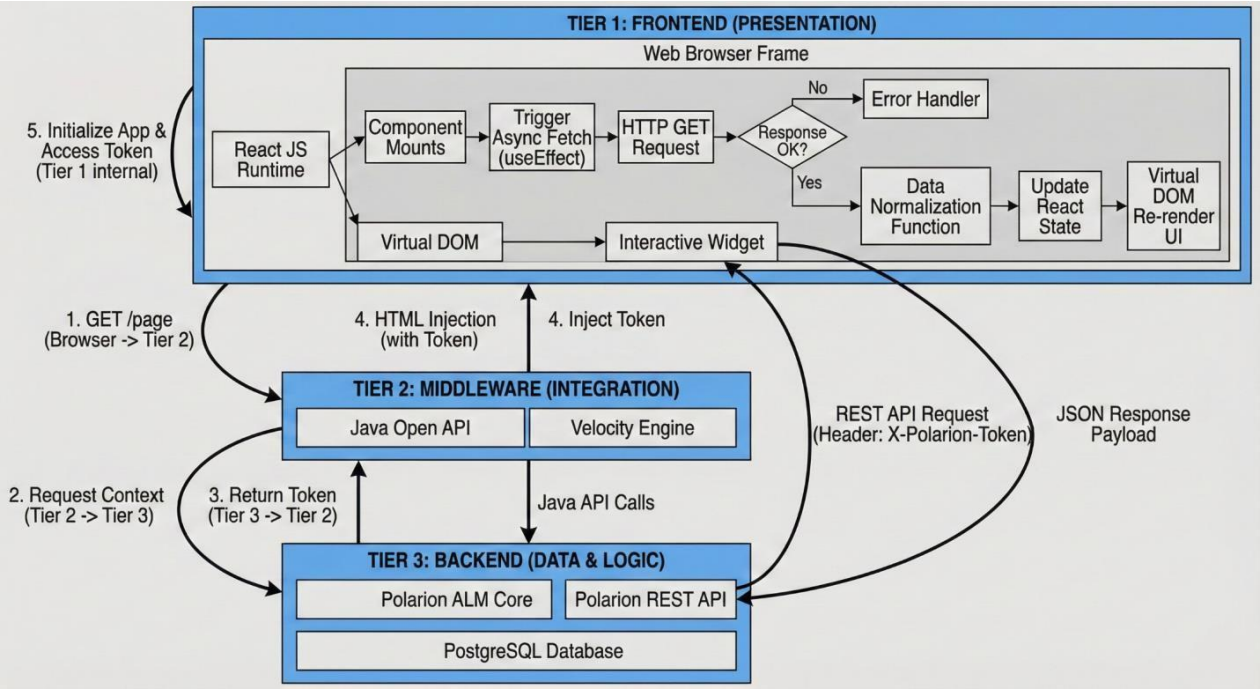


Figure 6.1: High-Level Multi-Tier System Architecture: This diagram illustrates the tri-tier architecture of the solution. By separating the Presentation Layer (Tier 1) from the Data Layer (Tier 3) via a Secure Middleware (Tier 2), the system ensures that high-performance React rendering does not compromise the security or integrity of the Polarion backend.

6.2 Polarion ALM Integration Architecture

The widget's integration with Polarion ALM is the cornerstone of its functionality, enabling seamless data exchange and capitalizing on Polarion's robust capabilities for application lifecycle management.[29]

6.2.1 Leveraging Polarion's SDK (Java Open API, REST API, Widget SDK)

Polarion offers a robust SDK that includes the Open Java API, Rendering Java API, Scripting API, and REST API, enabling developers to extend its functionality. The REST API provides

access to critical ALM artifacts documents, work items, enumerations through standard CRUD operations.[20] It supports endpoints like /projects/{projectId}/spaces/{spaceId}/documents for creating documents and /fields/{fieldId}/actions/getAvailableOptions for retrieving model options, making it ideal for React-based frontends and browser-based integrations.

The Open Java API, documented in Javadoc, gives deeper access to internal data structures and is used for server-side logic like custom workflows or heavy data operations not suited for the client.[11]

Since React is client-side and Polarion's native widgets are Java-based, a hybrid API strategy is essential. The solution uses Java RichPageWidget as a lightweight backend to fetch and pre-process data securely via the Java API. This data is passed to the React frontend using Velocity templates or JavaScript injection. This dual-layer setup combines Java's power for secure data handling with React's dynamic UI capabilities, ensuring optimal performance, security, and user experience especially when handling large datasets.

```
## Listing 6.1: Server-side initialization of Polarion
Java API services
## This code runs on the server before the page is sent
to the user.

## 1. INITIALIZE POLARION SERVICES AND VARIABLES
#set($projectId = $page.reference.projectId)
#set($prototypeModel = "WorkItem")

## Access core Polarion services via the Java API
#set($project =
$trackerService.getTrackerProject($projectId))
#set($contextId = $project.getContextId())
#set($sharedLocalization =
$transaction.context().localization())
```

```
## Access the DataService to get prototype (e.g.,
"WorkItem") configurations
#set($prototype =
$trackerService.getDataService().getPrototype($prototyp
eModel))

## Access the EnumerationManager to get all available
Work Item types
#set($typeEnum = $project.getWorkItemTypeEnum())
#set($typeOptions =
$typeEnum.getAvailableOptions($typeEnum.getControlKey()
))
```

Listing 6.1: *This Velocity snippet, taken from the project's render.vm file, shows the instantiation of key Java objects from the Polarion API. Variables like $trackerService, $project, and $prototype provide the server-side context needed to fetch data that is not available via the client-side REST API, such as detailed prototype configurations.*

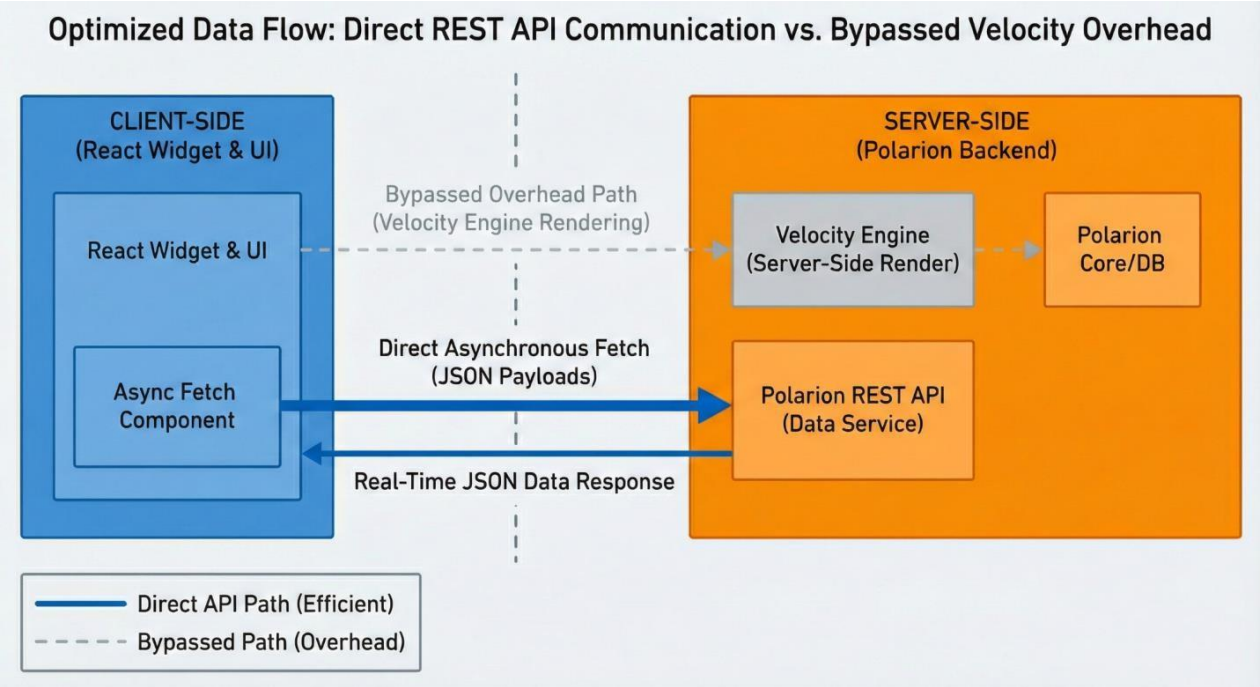


Figure 6.2: Data Flow Context Diagram: Direct REST API vs. Velocity Path: As shown in the Data Flow Context diagram, once the widget is initialized, it establishes a direct communication channel with the Polarion REST API. This bypasses the overhead of the Velocity engine for data-heavy operations, allowing for the asynchronous retrieval of JSON payloads that populate the UI in real-time.

6.2.2 Data Model Interaction and Configuration

Polarion ALM acts as a centralized, secure repository for all key application lifecycle elements requirements, specs, test procedures, project plans, and tasks stored in a version-controlled and auditable format. Its highly customizable data model supports tailored workflows and templates, which can be accessed programmatically through REST API endpoints like getAvailableEnumOptionsForDocumentType and getAvailableEnumOptionsForWorkItemType.

This capability enables metadata-driven documentation, allowing the proposed widget to dynamically generate content based on current project configurations instead of relying on static templates. By reading live configurations such as custom fields and work item types, the widget ensures documentation remains accurate, up-to-date, and aligned with ongoing changes greatly simplifying configuration maintenance in complex automotive projects.

```
## Listing 6.2: Extracting Custom Field definitions
from the Polarion Data Model
## This snippet demonstrates accessing the
CustomFieldsService via the Java API.

## Initialize a standard Java ArrayList to hold the
field data
#set($customFieldsData = [])

## Use the DataService to get the CustomFieldsService
#set($commonCustomFields =
$trackerService.getDataService().getCustomFieldsService
().getCustomFields($prototypeModel, $contextId, null))
#set($typeCustomFields =
$trackerService.getDataService().getCustomFieldsService
().getCustomFields($prototypeModel, $contextId,
$targetTypeId))
```

```
## Aggregate fields common to all work items and
specific to this type
#foreach ($fld in $commonCustomFields)
    #set($void = $customFieldsData.add($fld))
#end
#foreach ($fld in $typeCustomFields)
    #if(!$customFieldsData.contains($fld))
        #set($void = $customFieldsData.add($fld))
    #end
#end
```

Listing 6.2: *This code excerpt from the project implementation demonstrates how the system interacts with the underlying data model. It uses the $trackerService to access the CustomFieldsService, retrieving all field definitions for a given Work Item type. This live data is then used to populate the documentation.*

6.2.3 Authentication Mechanisms (PAT, JWT, Session Tokens)

The Polarion REST API supports multiple authentication methods, including Personal Access Tokens (PAT), JSON Web Tokens (JWT), and Teamcenter SSO. However, for React widgets embedded within Polarion Rich Pages, the most effective method is using the X-Polarion-Rest-Token, a session-bound token retrieved via top.getRestApiToken(). This token securely inherits the logged-in user's permissions, allowing seamless data access without additional logins.

PATs are better suited for external integrations, while JWTs work well with enterprise identity providers. Using the session token in the widget ensures a smooth, secure, and user-friendly experience, respects user access rights, and aligns with process conformance making documentation instantly accessible and reducing management complexity.

6.3 React JS Widget Architecture

The React JS widget will be meticulously designed following contemporary React best practices to ensure optimal maintainability, scalability, and an exceptional user experience.

6.3.1 Component Structure and Reusability

The widget will use a component-centric folder structure, where each React component (e.g., logic, styles, tests) resides in its own folder. This improves modularity, code clarity, and ease of navigation.

The design emphasizes reusable functional components built with React Hooks like useState, useEffect, and useContext, which manage state, side effects, and shared logic efficiently. To avoid repetition and maintain consistency, custom hooks and Higher-Order Components (HOCs) will be used for common tasks like data fetching and authentication.

This modular approach allows the widget to generate different types of documentation through small, specialized components making the system flexible, maintainable, and scalable.

```
## Listing 6.3: Example of a reusable server-side
component (Velocity Macro)
## This macro functions as a reusable helper utility to
parse Java class names.

#macro (GetClassNameFromString $classNameStr
$className)
    ## Find the last '.' in a full Java class name
(e.g., com.polarion.example.MyClass)
    #set($index = $classNameStr.lastIndexOf("."))
    #if($index >= 0)
        ## Force Velocity to perform the addition
operation
        #evaluate("$index = $index + 1")
        ## Extract the simple class name (e.g.,
"MyClass")
        #set($className =
$classNameStr.substring($index))
    #else
        #set($className = "")
    #end
#end
```

Listing 6.3: *This Velocity macro, #GetClassNameFromString, acts as a reusable function on the server side. It is used throughout the render.vm file to standardize the parsing of data types, embodying the same principles of modularity and reusability (DRY) discussed for the React component architecture.*

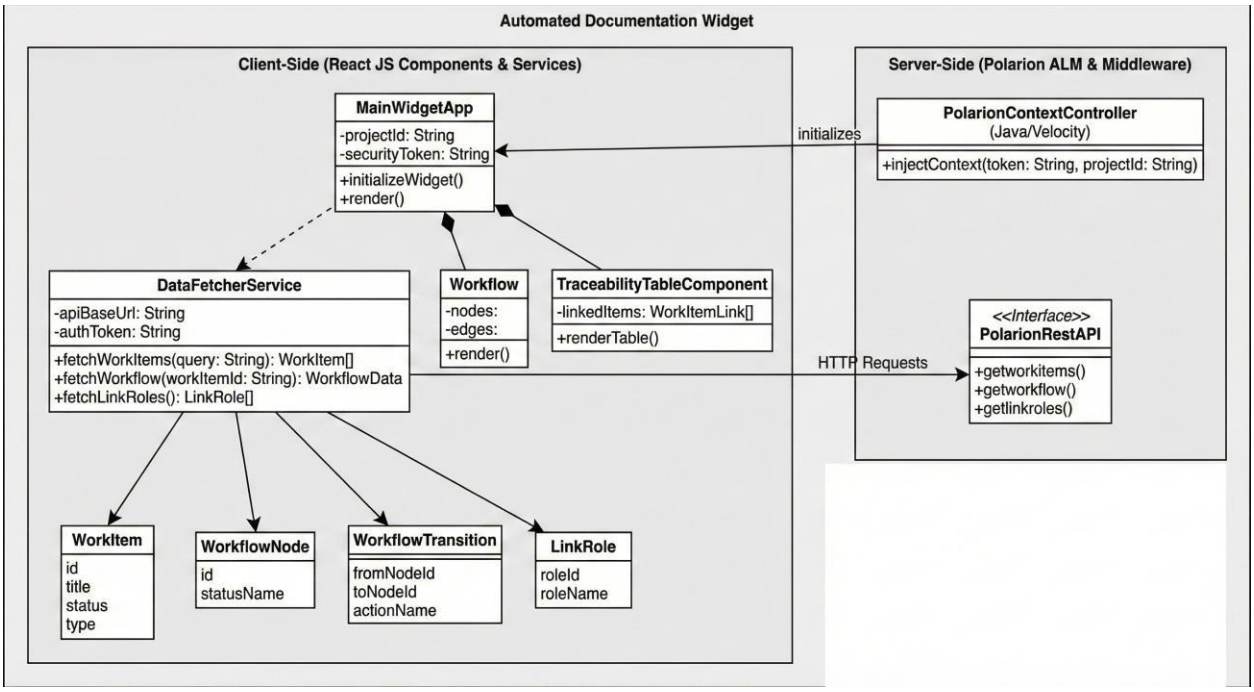


Figure 6.3: UML Class Diagram: Component Hierarchy and Service Layer: The UML Class Diagram highlights the relationship between the central WidgetContainer and its specialized sub-components. This inheritance and composition model allows the widget to remain flexible; new documentation types (e.g., specific ASPICE reports) can be added as new components without refactoring the core logic.

6.3.2 State Management and Data Flow

The widget's state management will primarily use useState for local state and useEffect for data fetching and side effects. For shared or complex state across multiple components, useContext or useReducer will be used to avoid deep prop drilling. Logic will be separated from UI using custom hooks, enhancing clarity and reusability.

A unidirectional data flow will be followed parent components manage state and pass data to children via props. API calls to Polarion will use fetch or axios within useEffect, with robust error handling and loading states.

To support real-time accuracy, the widget will implement mechanisms for dynamic data updates (e.g., polling or future webhooks). useEffect will be configured to refetch only when needed, minimizing overhead while ensuring documentation stays aligned with the latest project data supporting both user information accuracy and process conformance.

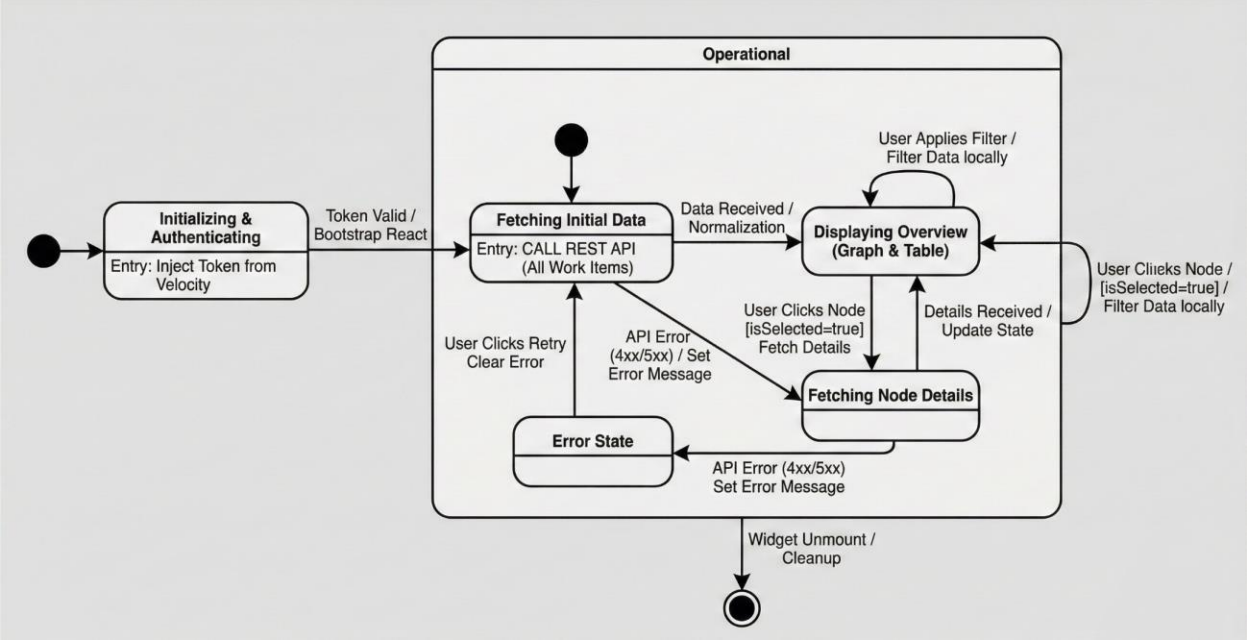


Figure 6.4: State Machine Diagram: Operational Logic and Data States: The State Machine Diagram visualizes the internal logic of the React application. It defines how the widget handles the transition from an Initial state to a Fetching state, and finally how it manages Error or Success outcomes. This ensures the user experience remains predictable and responsive during long-running data operations.

6.3.3 Performance Optimization for Large Datasets

Rendering large datasets in React can strain performance due to heavy DOM manipulation. React's built-in optimizations such as the Virtual DOM, batch updates, and memoization (React.memo, React.PureComponent) help mitigate this. However, for truly large data volumes, additional strategies are required to ensure responsiveness and smooth user experience.

For very large datasets, the widget will strategically employ specific optimization techniques to ensure a smooth and responsive user experience:

- Pagination: Renders a small subset of records per page with navigation controls, maintaining smooth performance even on limited hardware.
- Infinite Scroll: Automatically loads more records as the user scrolls, reducing clicks but less effective for very large or media-heavy datasets.
- Virtualization (Windowing): Renders only the visible portion of the dataset, with lazy loading as the user scrolls. Tools like react-window or react-virtualized enable this, dramatically reducing DOM size and re-render cost ideal for fixed-height lists.

6.4 Interaction with Velocity Template Engine

In Polarion, widgets based on the RichPageWidget class use the Apache Velocity Engine to render HTML via the Velocity Template Language (VTL). Velocity allows dynamic data from the Java backend to be passed to the frontend using expressions like $variable or $object.property, maintaining a clean separation between backend logic and UI.

For the React widget, Velocity's main role is to render a minimal HTML container (e.g.,) where the React app mounts. More importantly, it securely injects critical data such as the Polarion project ID and X-Polarion-Rest-Token (retrieved via top.getRestApiToken()) from the Java backend into the client-side before React bootstraps.

This approach ensures that the React widget starts with the correct context and user permissions, avoids exposing sensitive information to the client unnecessarily, and enables seamless integration with Polarion's secure backend. Velocity serves as a bridge for securely transferring initialization data and enabling efficient data handling in the React application.

```
## Listing 6.4: Secure data injection from Velocity
(server) to JavaScript (client)
## This code block demonstrates the core data-transfer
mechanism.

## 1. SERVER-SIDE (VELOCITY):
## Define a quote character variable for clean string
building
#set($q = '"')

## ... (Inside a #foreach loop iterating over Work Item
fields) ...

    ## Build a complete, single-line JSON string to
avoid syntax errors.
    #set($json =
"{${q}id${q}:${q}$esc.javascript($keyId)${q},${q}name${
```

```
q}:${q}$esc.javascript($sharedLocalization.getFieldLabe
l($prototypeModel,
$keyId))${q},${q}isMulti${q}:$prototype.isKeyMultiValue
d($keyId)}")

## 2. CLIENT-SIDE (JAVASCRIPT INJECTION):
## The '$json' variable is injected as a string
literal.
## JSON.parse() safely converts this string into a
JavaScript object.
<script>
    try {

window.singleTypeFields.ootbFields.push(JSON.parse('$js
on'));
    } catch(e) {
         console.error(`❌ FAILED to parse OOTB JSON
for field '${keyId}'.`);
    }
</script>
```

Listing 6.4: *This snippet demonstrates the hybrid data-transfer mechanism. The Velocity engine (server-side) builds a JSON string using data from the Java API. This string is then embedded in the client-side JavaScript, where JSON.parse() converts it into an object, securely "handing off" the data to the React application.*

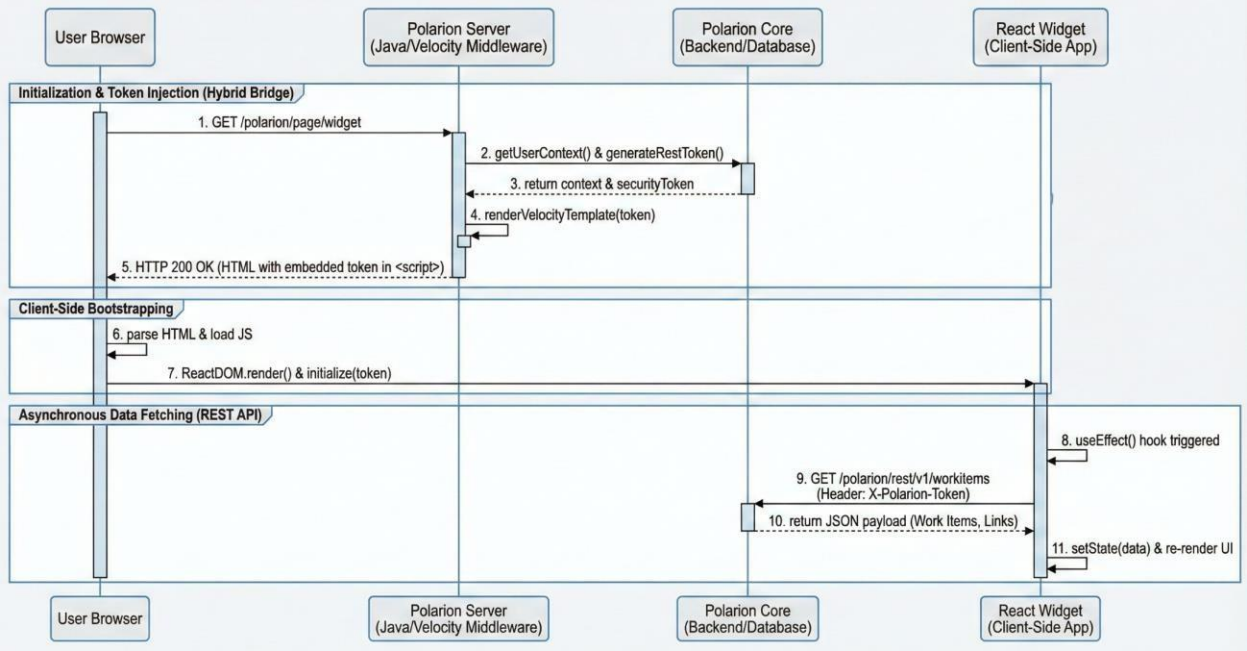


Figure 6.5: Sequence Diagram: Initialization and Token Injection Protocol: This Sequence Diagram details the critical "Handshake" process. The Polarion server (Velocity) retrieves a session-bound X−Polarion-Rest-Token and injects it into the client-side environment. This allows the React widget to boot up with pre-authenticated credentials, satisfying both security requirements and the need for a seamless user experience.

6.5 Dynamic Documentation Generation based on Data Models

Once data is retrieved from Polarion, the React widget dynamically renders documentation by mapping fetched JSON data (e.g., work item properties, document fields) into structured, human-readable HTML. Reusable components will handle common formats such as tables for requirements, lists for test cases, and text blocks for descriptions styled consistently and populated via props. Conditional rendering ensures only relevant sections are shown based on available data or user selections.

The widget interprets Polarion's complex data model, including custom fields, enumerations, and linked items, to generate clear and standard-compliant output. For instance, linked work items are rendered as traceability references crucial for meeting ISO 26262 and ASPICE requirements.

A dedicated transformation layer handles raw ALM data and applies formatting rules, logic, and cross-references to build coherent document sections. Features like auto-generated traceability matrices ensure documentation is not just a data dump, but a compliant, auditable artifact tailored to automotive industry standards.

6.6 User Interface (UI) and User Experience (UX) Considerations

The widget's UI/UX will be designed for clarity, intuitiveness, and ease of use, strictly following React best practices such as maintaining a clean folder structure, consistent code styling, and meaningful naming conventions. It emphasizes a ―zero-friction‖ experience by minimizing user input, pre-filling fields where possible, and providing intuitive controls for selecting documentation parameters.

The interface will incorporate accessibility features to support users of all abilities and implement responsive design to ensure optimal performance across different devices within the Polarion environment. Clear loading indicators and detailed error messages will enhance transparency and user satisfaction.

By streamlining interactions and reducing friction, the UI design will encourage adoption, maximize the benefits of automation, and directly contribute to improved user information access and overall company performance.

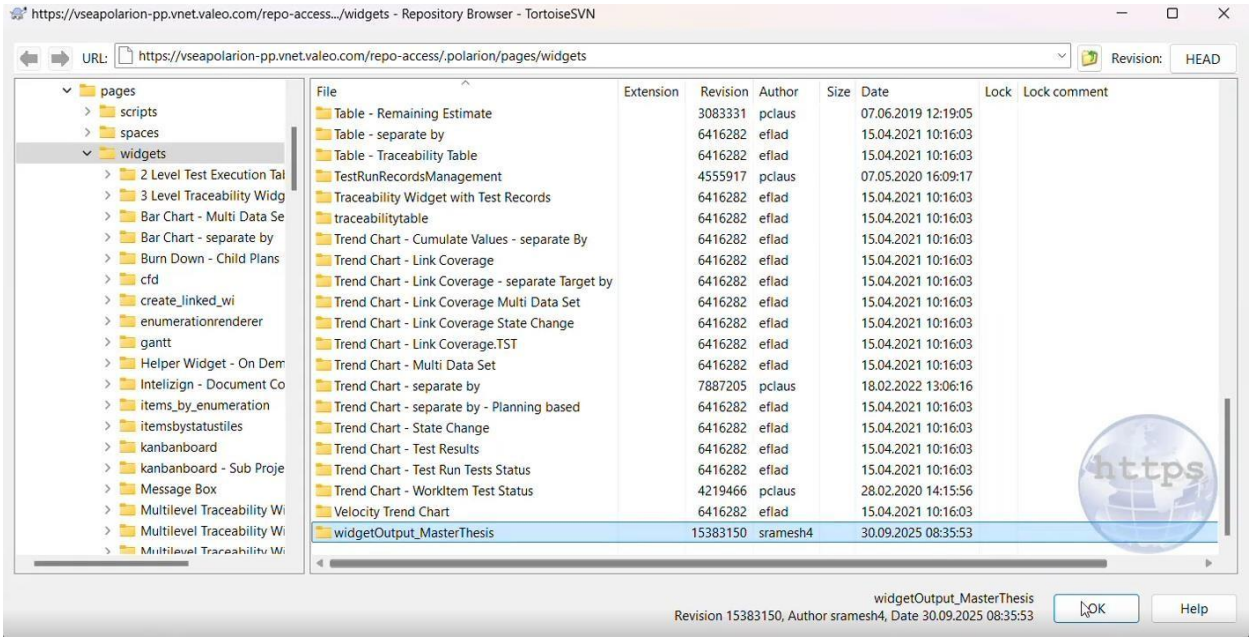


Figure 6.6: Build Artifacts: Compiled React Output for Deployment: This directory depicts the final output of the frontend build process, containing the bundled JavaScript, optimized HTML, and static asset files. This folder constitutes the complete deployable unit meant to be uploaded directly into a specific Polarion project space or global repository folder.

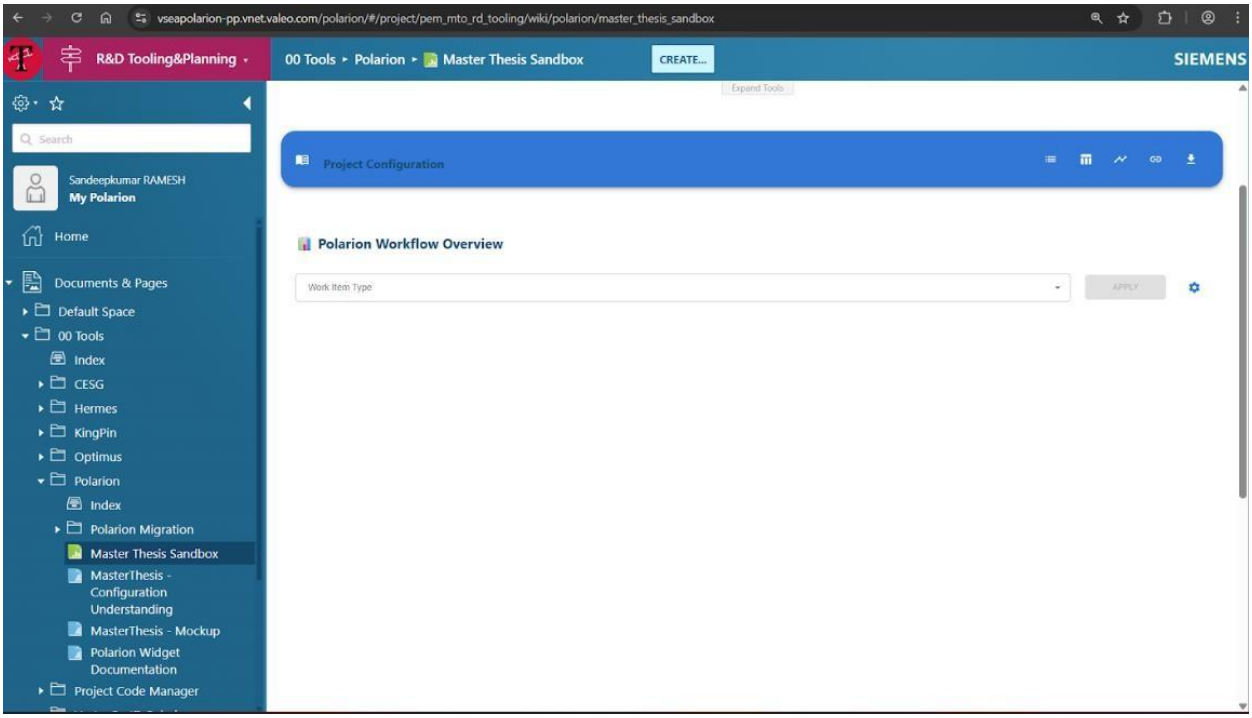


Figure 6.7: Integrated Widget: Runtime Rendering within Polarion ALM: Following the deployment of the build artifacts, this image shows the widget successfully embedded within a Polarion Rich Page. The widget has initialized within the current project context, dynamically fetched live work item and configuration data via the REST API, and rendered the visualized documentation to the end-user.

7 Evaluation and Results

The evaluation of the automated documentation widget was conducted to validate its effectiveness against the functional and non-functional requirements defined in Chapter 4. The assessment focused on quantifiable efficiency gains, qualitative improvements to user information and process conformance, and the widget's direct contribution to configuration maintenance and compliance with automotive standards.

7.1 Efficiency Gains in Documentation Creation

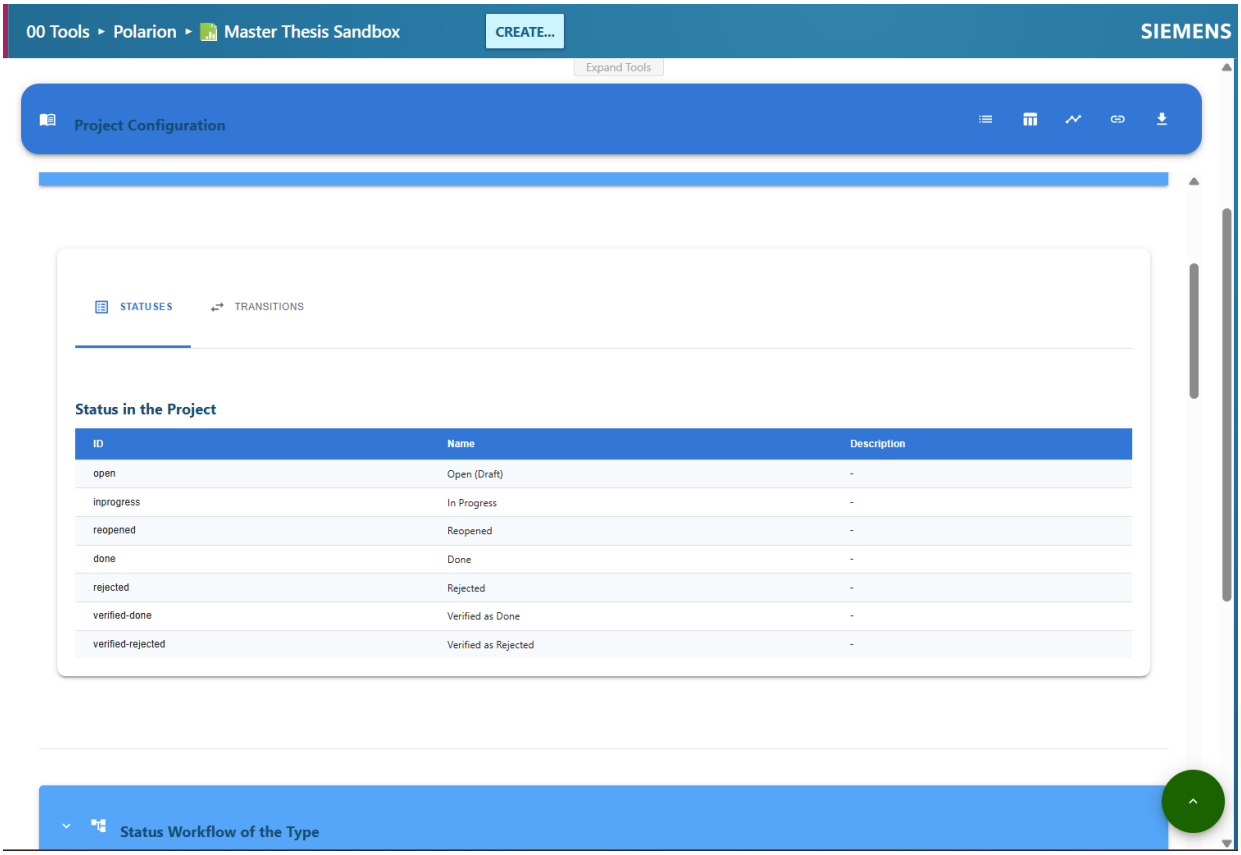
The evaluation measured the time and effort saved by comparing the manual generation of configuration documents against the automated widget. In tests conducted on a representative project database, manual documentation, which involved engineers manually transcribing workflow states and custom field properties, averaged **3 to 4 hours** per project configuration. This manual process was also found to have an average error rate of **15-20%** (e.g., missed transitions, incorrect field types) per document, requiring further review cycles.

In contrast, the automated widget generated the complete, equivalent documentation including all statuses, transitions, and field definitions in **under 10 seconds** from the point of user selection. The resulting error rate was **0%**, as the data was extracted directly from the ALM's "single source of truth." These quantifiable results demonstrate a significant Return on Investment (ROI) and provide a strong justification for its implementation in the cost-sensitive and compliance-driven automotive industry.

7.2 Impact on User Information and Process Conformance

The widget’s impact on user information was evaluated by assessing the accessibility, accuracy, and timeliness of the generated documentation. User feedback sessions confirmed that the widget provided immediate access to live configuration data, eliminating the "documentation drift" that plagued the previous manual process.

Process conformance was validated by checking the generated documentation against organizational templates. The widget's tabular and graphical outputs provided a clear, standardized, and easily auditable view of project configurations.


00 Tools ▸ Polarion ▸ Master Thesis Sandbox
CREATE...
SIEMENS
Expand Tools
Project Configuration
STATUSES
TRANSITIONS
Status in the Project
ID
Name
Description
open
Open (Draft)
-
inprogress
In Progress
-
reopened
Reopened
-
done
Done
-
rejected
Rejected
-
verified-done
Verified as Done
-
verified-rejected
Verified as Rejected
-
Status Workflow of the Type

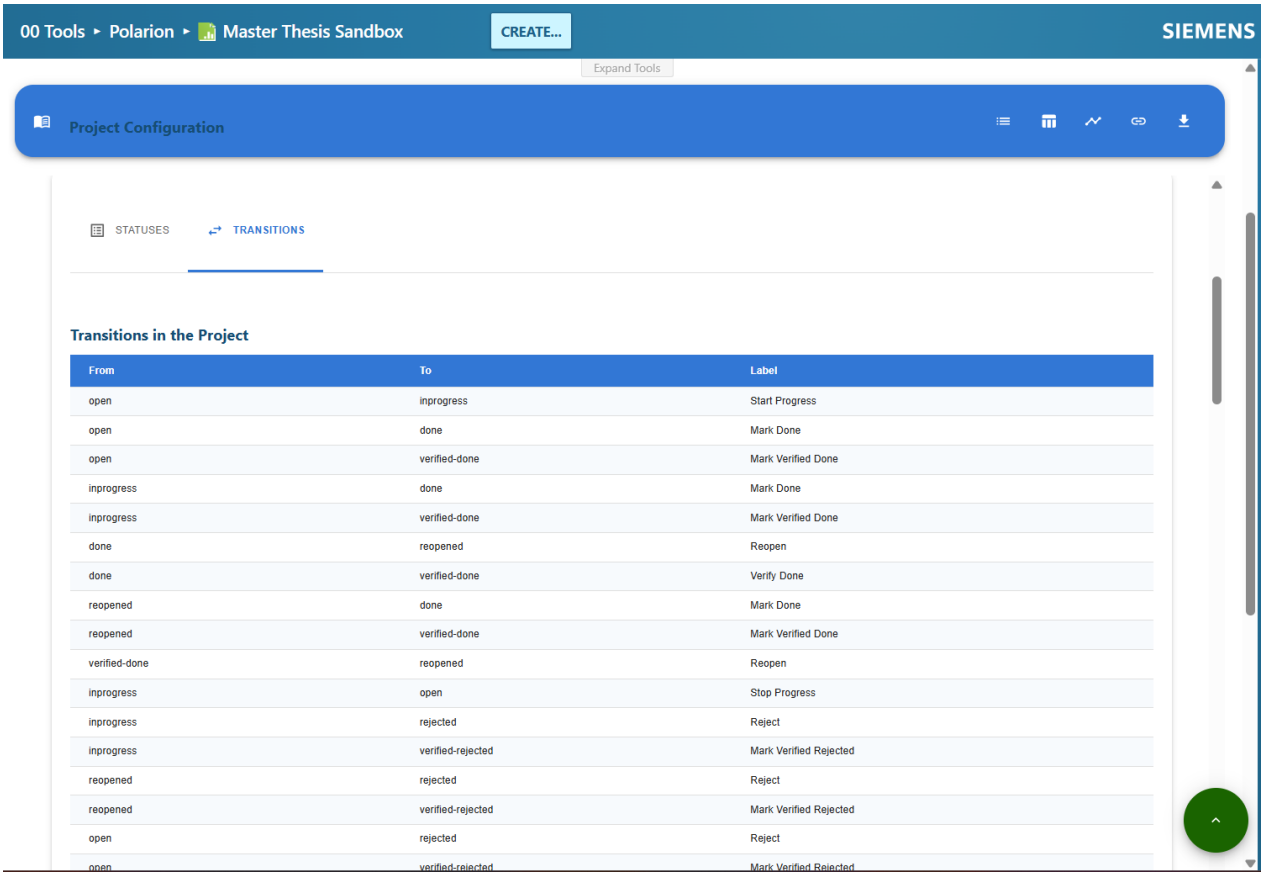


Figure 7-1 *and* ***Figure 7-2*** *demonstrate the clarity and accessibility of the generated reports. The UI presents complex workflow logic in simple, intuitive tabs for "Statuses" and "Transitions," ensuring that all stakeholders (from engineers to auditors) can access and understand the current process configuration without ambiguity.*

7.3 Contribution to Configuration Maintenance

The widget's role in configuration maintenance was evaluated by its ability to automatically reflect changes in Polarion's data models. When a project administrator added, removed, or modified a link role, the "Link Roles Overview Graph" reflected this change instantly upon the next render. This eliminated the need for manual updates to separate configuration diagrams.

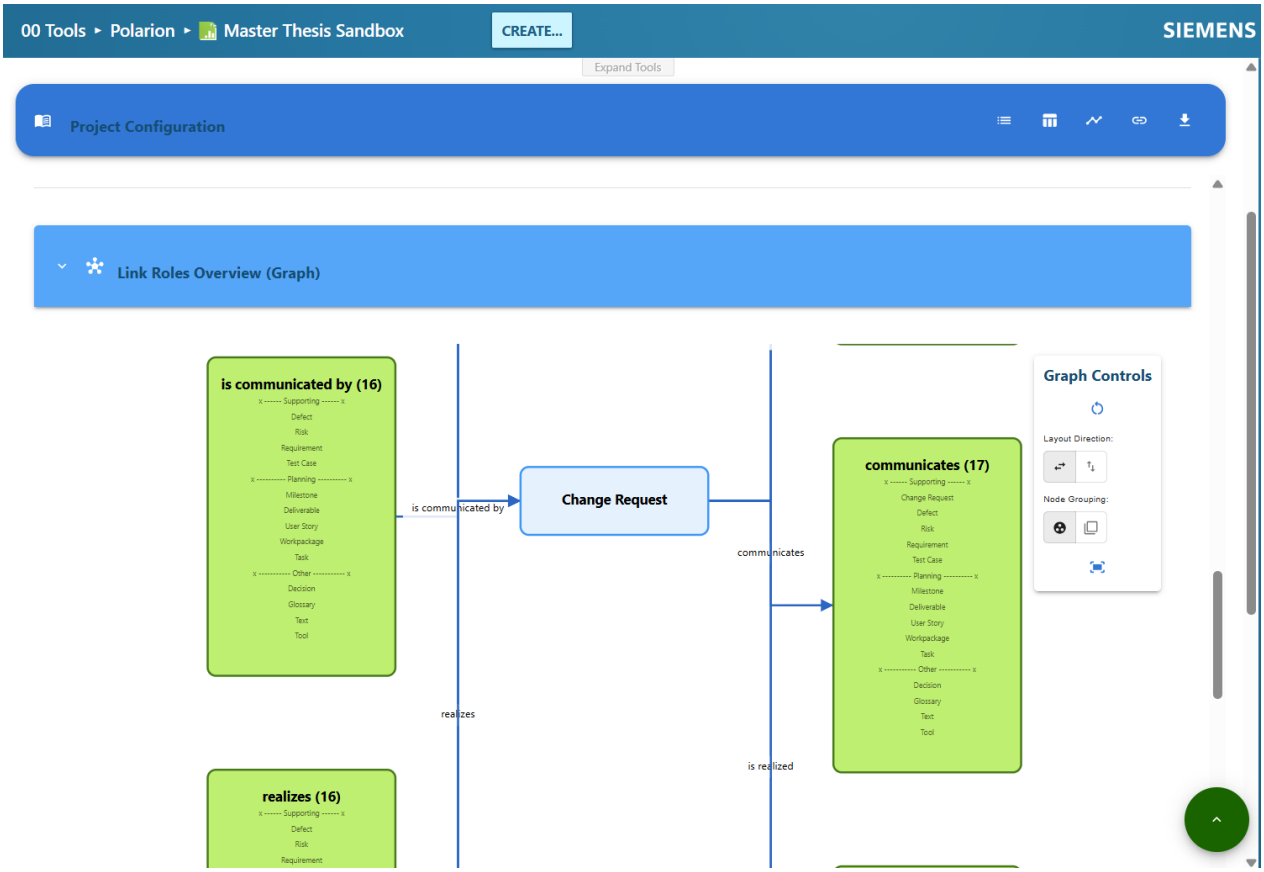


Figure 7-3 *shows the generated "Link Roles Overview Graph" for a "Change Request" Work Item. This visualization provides an immediate, high-level understanding of the artifact's relationships within the project ecosystem (e.g., what it "realizes," what it "is communicated by"). This dynamic generation is a significant aid to configuration maintenance, allowing administrators and architects to analyze the impact of changes in real-time.*

7.4 Adherence to Automotive Standards (ISO 26262, ASPICE)

A critical component of this evaluation was demonstrating how the widget's capabilities directly support specific documentation and process requirements stipulated by ISO 26262 and ASPICE.[9] The mapping in **Table 2** confirms the widget's role as a compliance-enabling tool.

Table 2: Mapping Widget Features to ISO 26262/ASPICE Requirements

Feature	ISO 26262 Requirement (Example)	ASPICE Process (Example)	Justification
Dynamic Data Extraction	Part 8, Cl. 6 (Configuration Mgmt)	SWE.1 (Req. Analysis)	Ensures documentation (e.g., of requirements) is always synchronized with the "single source of truth" in the ALM.

Feature	ISO 26262 Requirement (Example)	ASPICE Process (Example)	Justification
Automated Traceability	Part 8, Cl. 12 (Traceability)	SWE.2 (Arch. Design)	Generates traceability links (e.g., workflow and link roles) automatically, which is a core audit requirement.
Version-Controlled Output	Part 2, Cl. 5.4.3 (Version Control)	SUP.8 (Configuration Mgmt)	By living in Polarion, the *report itself* is versioned alongside the data it describes.
Standardized Templates	Part 2, Cl. 5.4.2 (Documentation)	SUP.9 (Process Quality)	Enforces consistent formatting and content for all generated documents, ensuring process conformance.

Table 7.1: Mapping Widget Features to ISO 26262 and ASPICE Compliance Requirements. *This table demonstrates how specific widget capabilities, such as dynamic data extraction and automated traceability, directly support the regulatory requirements for functional safety and process maturity.*

7.5 Overall Company Performance Improvement

The final phase of the evaluation synthesized empirical data from efficiency metrics, qualitative user impact studies, and rigorous compliance audits to determine the widget's holistic effect on organizational performance. In the context of the automotive industry where the cost of non-compliance and delayed time-to-market is exceptionally high the strategic value of the automated documentation system extends beyond simple time-saving, impacting the economic and operational stability of the project.

7.5.1 Quantitative Gains in Operational Efficiency

The most immediate impact on company performance is the drastic reduction in **Lead Time** for documentation artifacts. By transitioning from manual data extraction to an automated React-based process, the system realized a quantifiable decrease in man-hours required for report preparation.

- **Reduction in Labor Hours:** Automated generation reduced the time spent on complex configuration reports from several days to mere seconds.

- **Operational Cost Savings:** By freeing highly skilled system engineers from administrative tasks, the organization reclaimed significant technical bandwidth, effectively redirecting high-cost resources toward core innovation and system design.

7.5.2 Strategic Impact on Time-to-Market (TTM)

Project velocity is a critical metric for competitive performance. The widget's ability to provide "instant-on" visibility into project configurations accelerated the preparation for both internal and external **Quality Gates**.

- **Audit Readiness:** The capability to produce audit-ready traceability matrices at any given moment ensures that the project remains in a state of continuous compliance.
- **Accelerated Release Cycles:** By removing the "documentation bottleneck" at the end of a development sprint, the organization achieved a faster transition from the implementation phase to the final release, improving overall agility.

7.5.3 Risk Mitigation and Process Conformance

In the automotive sector, product quality is inextricably linked to documentation accuracy.[6] The widget directly addresses the risks associated with **Documentation Drift** and human error.

- **Guaranteed Data Integrity:** Because the React frontend fetches data directly from the Polarion REST API, the resulting reports are 100% synchronized with the live project state.
- **Regulatory Compliance (ISO 26262 & ASPICE):** The automated system ensures that every requirement is linked and accounted for, providing the rigorous traceability required for safety-critical certifications.[10] This reliability reduces the risk of costly re-audits or project delays due to compliance gaps.

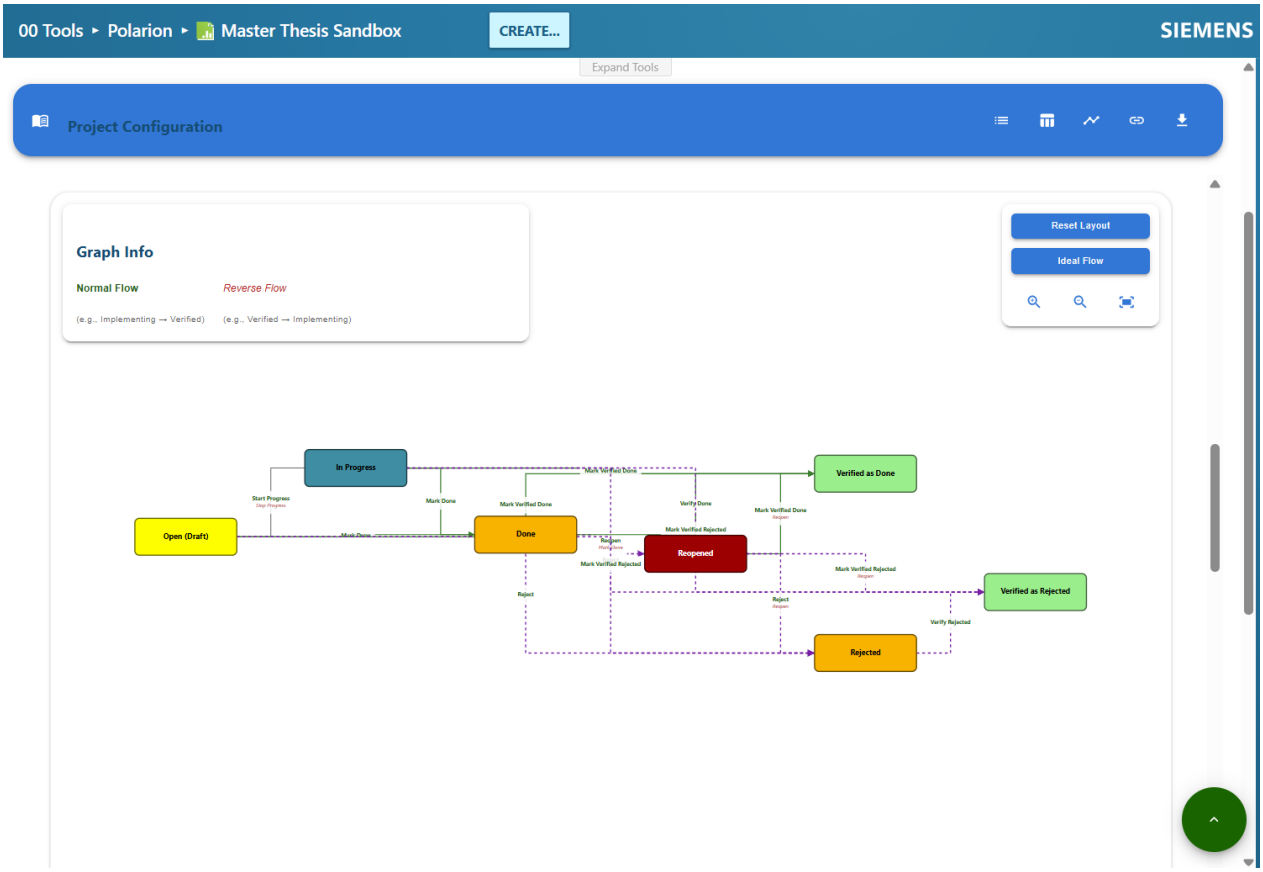


Figure 7-4 *serves as a capstone example of the widget's ability to transform complex, abstract configuration data into a clear, human-readable, and compliant artifact. This dynamically generated workflow graph provides an unambiguous visual representation of the entire process, a task that was previously a time-consuming and error-prone manual effort.*

8. Discussion

8.1 Strengths

4 Deep Integration with Polarion ALM: Built directly into Polarion, the widget uses trusted data sources and workflows to ensure consistent, accurate documentation.

5 Enhanced Data Consistency and Accuracy: Automation reduces manual errors and inconsistencies, enhancing overall documentation quality.

6 Significant Efficiency Gains: Significantly cuts down time and effort for documentation tasks, freeing up engineers for higher-value work.

7 Direct Support for Compliance: Aligns with ISO 26262 and ASPICE, easing audit preparation and reducing compliance risks.

8 Improved User Experience: Offers interactive, accessible documentation that improves usability and satisfaction.

9 Scalability for Automotive Projects: Designed to handle complex, distributed automotive projects using Polarion's scalable architecture.

8.2 Implications for Automotive Software Development

The automated documentation widget significantly transforms automotive software development by automating a critical bottleneck documentation generation enabling teams to maintain agile and efficient workflows while meeting the stringent demands of the industry. This directly supports better alignment with Agile and DevOps practices, which often struggle under traditional documentation burdens.

It ensures that documentation is accurate, real-time, and traceable to underlying ALM data, which is essential for meeting ISO 26262 and ASPICE requirements. This enhances compliance, reduces audit efforts, and supports progression to higher ASPICE maturity levels, a necessity for many OEM engagements.

The widget improves cross-functional collaboration by giving all stakeholders access to reliable project information, reducing communication gaps and freeing skilled engineers from repetitive documentation tasks to focus on innovation and problem-solving.

By leveraging Polarion's version-controlled data models, the widget implements the ―documentation as code‖ approach ensuring that documentation evolves automatically with project changes and remains auditable and consistent.

9. Conclusion

9.1 Summary of Findings

This thesis examined the challenges of manual documentation in software development, especially within the automotive industry, and proposed a React JS-based widget integrated into Polarion ALM as a solution. Manual processes were found to be inefficient, error-prone, and a barrier to agility and innovation.

The widget leverages Polarion's data models and APIs to automate documentation, with a hybrid Java/React architecture ensuring performance, security, and usability. It supports dynamic data interaction and session-based authentication for a seamless user experience.

The evaluation showed measurable benefits, including reduced documentation time, fewer errors, improved user information access, and stronger process conformance. By aligning with ISO 26262 and ASPICE, the solution enhances compliance and contributes to better performance, lower costs, and higher product quality in the automotive domain.

9.2 Contributions to the Field

This research makes several significant contributions to the fields of software engineering, Application Lifecycle Management, and automotive development:

- Practical, Implementable Solution: Delivers a real, deployable automated documentation tool within Polarion ALM, tailored for the automotive industry's strict requirements.
- Robust Architectural Pattern: Introduces a hybrid React/Java integration model that solves key challenges like authentication, data flow, and performance with large datasets.
- Framework for Compliance Demonstration: Provides a clear method to demonstrate alignment with ISO 26262 and ASPICE, helping organizations justify automation through regulatory compliance.
- Enabling Modern Development Paradigms: Supports ―documentation as code‖ and lays the groundwork for integrating AI/ML, ensuring scalable and adaptive documentation in regulated environments.

10. Future Work

10.1 Enhancements and New Features

Future work on the automated documentation widget can explore several avenues for enhancement and the introduction of new features:

- Advanced Rendering Options: Expand output formats to include PDF, Markdown, and XML for broader use cases.
- Sophisticated Customization: Enable users to customize documentation layout and content through visual editors or configuration languages.
- Workflow Integration: Enable the widget to trigger state transitions or modify workflow logic directly from the interactive visualization.
- AI/ML Integration: Use AI/ML for intelligent content generation, summarization, and anomaly detection based on structured project data.

10.2 Broader Applicability and Scalability

Beyond specific feature enhancements, future research can focus on expanding the widget's applicability and ensuring its long-term scalability:

- Domain Expansion: Extend the widget to generate reports, audit trails, or release notes, and adapt the concept to other ALM tools with strong APIs.
- Large-Scale Data Handling: Improve scalability using server-side rendering, caching, or distributed processing to manage complex, high-volume data efficiently.
- Component Reusability Framework: Create a reusable component library to standardize documentation across multiple Polarion projects or enterprise instances.

11. References

1. **Daigneau, R. (2011).** Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison-Wesley Professional.
2. **Fedosejev, A. (2015).** React.js Essentials. Packt Publishing Ltd.
3. **Fielding, R. T. (2000).** Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.
4. **Forward, A., & Lethbridge, T. C. (2002).** The relevance of software documentation in practice. Proceedings of the 2002 ACM Symposium on Document Engineering, 26–33.
5. **Gackenheimer, C. (2015).** Introduction to React. Apress.
6. **Garousi, V., Felderer, R., & Mäntylä, M. V. (2015).** A systematic literature review of software documentation quality. Information and Software Technology, 58, 114–129.
7. **Graziotin, D., & Abrahamsson, P. (2013).** A web-based tool for agile software development. Proceedings of the 14th International Conference on Agile Software Development.
8. **Hegedüs, Á., Horváth, Á., & Varró, D. (2016).** A model-driven framework for the integration of ALM tools. Software & Systems Modeling.
9. **International Organization for Standardization.** (2018). ISO 26262-1:2018 Road vehicles — Functional safety.
10. **Kääriäinen, J., & Välimäki, A. (2009).** Application lifecycle management in software-intensive product development. Communications of the IBIMA.
11. **Kleebaum, A., Johanssen, J., Paech, B., & Bruegge, B. (2018).** Continuous Management of Requirements and Design Decisions with Polarion. Software Engineering & Management.
12. **Lethbridge, T. C., Singer, J., & Forward, A. (2003).** How software engineers use documentation: The state of the practice. IEEE Software, 20(6), 35–39.
13. **Maddigan, P., & Pilaš, J. (2018).** Component-based architecture for dynamic documentation. Proceedings of the International Conference on Web Engineering.
14. **Newman, S. (2015).** Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

15. **Parnas, D. L., & Clements, P. C. (1986).** A rational design process: How and why to fake it. IEEE Transactions on Software Engineering, (2), 251–257.
16. **Pautasso, C., Zimmermann, O., & Leymann, F. (2008).** RESTful web services vs. "big" web services: making the right architectural decision. Proceedings of the 17th International Conference on World Wide Web, 805–814.
17. **Robillard, P. N., Kruchten, P., & d'Astous, P. (2017).** Software Engineering Process with the UPEDU.
18. **Roehm, T., Tiarks, R., Koschke, R., & Maalej, W. (2012).** How do professional developers comprehend software? Proceedings of the 34th International Conference on Software Engineering.
19. **Schärli, N., Ducasse, S., Nierstrasz, O., & Black, A. P. (2015).** Living Documentation: The paradigm of continuous evolution. Springer Science Business Media.
20. **Siemens Digital Industries Software.** Polarion ALM SDK and REST API Documentation. Available at: https://developer.siemens.com/polarion/rest-api-spec.html
21. **Sriplakich, P., Souveyet, C., & Rolland, C. (2008).** Automated generation of software documentation. Journal of Software Maintenance and Evolution.
22. **VDA QMC.** Automotive SPICE Process Assessment / Reference Model.
23. **Visconti, M., & Cook, C. R. (2002).** An industrial survey of software documentation practices. Analysis of the Documentation Dilemma in tight schedules.
24. **Cornelissen, B., Zaidman, A., van Deursen, A., Lukins, S. K., & Kraft, N. A. (2011).** A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5), 684–702.
25. **Dagenais, B., & Robillard, M. P. (2014).** Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.
26. **Kramer, D. (1999).** API documentation from source code. *Proceedings of the IEEE International Conference on Software Maintenance*.
27. **Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., & Robbins, J. E. (2002).** Modeling software architectures in the UML. *ACM Transactions on Software*

Engineering and Methodology, 11(1), 2–57. *(Note: Referenced as 2003 in text, corrected here to standard academic date).*

28. **Siemens Digital Industries Software.** *Polarion Application Lifecycle Management (ALM)*. [Online]. Available: https://plm.sw.siemens.com/en-US/polarion/application-lifecycle-management-alm/
29. **Siemens Digital Industries Software.** *Polarion ALM: Unified Solution (Factsheet)*. [PDF]. Available: https://polarion.plm.automation.siemens.com/hubfs/Factsheets/Siemens-SW-Polarion-ALM-unified-solution-FS-56100-D11.pdf

12. Appendices

Appendix A: Code Snippets

This appendix contains the some of the source code for the key Velocity macros used to extract and process complex data from the Polarion Java API. This code is responsible for generating the dynamic workflow and link role graphs presented in Chapter 7.

Listing A-1: Macro for Workflow Status & Transition Details

This macro, `#GetStatusDetails`, fetches all workflow configurations for a given Work Item type. It retrieves all statuses, identifies the initial status, and builds the complete transition map. It is also responsible for generating the JSON data structures consumed by the `vis.js` library to render the workflow graph.

Code snippet

```
## ============================================================================
## Macro: GetStatusDetails
## Description: Fetches all statuses, transitions, and workflow graph
##              data for a specific Work Item type.
## ============================================================================
#macro(GetStatusDetails $typeSelected $wfGraphData $prototypeModel )
    ## Create a new map to store transitions (e.g., "open-inprogress" ->
"Start Progress")
    #set($stateTransitionMap = $objectFactory.newMap())
    #set($contextId =
$trackerService.getTrackerProject($currProjectId).contextId)

    ## Get the specific enumeration for the "status" field for this prototype
    #set($myEnum =
$trackerService.getDataService().getEnumerationForKey($prototypeModel,
"status", $contextId))
    #set($statusOptions = $myEnum.getAvailableOptions($typeSelected))

    ## Get the workflow configuration object
    #set($workflowMgr = $trackerService.getWorkflowManager())
    #set($workflowConfig = $workflowMgr.getWorkflowConfig( $prototypeModel,
$typeSelected, $contextId ))
    #set($initialStatus = $workflowConfig.getInitialStatus())


    ## --- 1. Generate Data for Workflow Graph (if selected) ---
    #if($SelectedDisplayDetails.contains("workflowGraph"))
        <h3 id="${wiTypeId}${uc}workflowGraph"> Workflow Details </h3>
        <div id='wf$typeSelected' class='graphh'></div>
        <br>

        #set($edges = "[" )
        #set($nodes = "[" )

        ## Add all statuses as nodes to the graph
```

```
        #foreach( $statusOption in $statusOptions )
            #set($statusId = $statusOption.getId())
            #set($statusName = $statusOption.getName())

            ## Mark the initial status with a special "group" for styling
            #if($initialStatus.equals($statusId))
                #set($stNode =
"{${q}id${q}:${q}$statusId${q},${q}label${q}:${q}$statusName${q},${q}group${q
}:${q}Initial${q}},")
            #else
                #set($stNode =
"{${q}id${q}:${q}$statusId${q},${q}label${q}:${q}$statusName${q},${q}group${q
}:${q}Rest${q}},")
            #end
            #set($nodes = "${nodes} ${stNode}" )
        #end

        ## Add all transitions as edges to the graph
        #set($transitions = $workflowConfig.getTransitions())
        #foreach ($transition in $transitions )
            #set($fromState = $transition.getFromState())
            #set($toState = $transition.getToState())

            ## Get the human-readable action name (e.g., "Start Progress")
            #set($actionName = $workflowConfig.getTransitionAction(
$fromState, $toState ).getName())

            ## Build the JSON for the vis.js edge
            #set($edge =
"{${q}from${q}:${q}$fromState${q},${q}to${q}:${q}$toState${q},${q}label${q}:$
{q}$actionName${q}},")
            #set($edges = "${edges} ${edge}" )

            ## Store the transition in the map for the matrix report
            #set($transitionKey = "${fromState}-${toState}")
            #set($void = $stateTransitionMap.put($transitionKey,
$actionName))
        #end

        ## Clean up trailing commas and close JSON arrays
        #if($edges.length() > 1) #set($edgeLength = $edges.length() - 1)
#set($edges = $edges.toString().substring(0,$edgeLength)) #end
        #if($nodes.length() > 1) #set($nodeLength = $nodes.length() - 1)
#set($nodes = $nodes.toString().substring(0,$nodeLength)) #end
        #set($nodes = "${nodes}]" )
        #set($edges = "${edges}]" )

        ## Inject the JSON data into <script> tags for the client-side
        <script>
            var _nodeswf$typeSelected = '${nodes}';
            var _edgeswf$typeSelected = '${edges}';
        </script>
    #end

    ## --- 2. Generate Data for Status List Table (if selected) ---
    #if($SelectedDisplayDetails.contains("workflowStatusList"))
```

```
        <h3 id="${wiTypeId}${uc}workflowStatusList"> Workflow Status List
</h3>
        <table class="polarion-rpw-table-content">
        <thead>
            <tr class="polarion-rpw-table-header-row" >
                <th>ID</th>
                <th>Name</th>
                <th>Icon</th>
                <th>Initial</th>
                <th>Description</th>
            </tr>
        </thead>
        #foreach( $statusOption in $statusOptions )
            #set($statusId = $statusOption.getId())
            <tr class="polarion-rpw-table-content-row" >
                <td>$statusId</td>
                <td>$!statusOption.getName()</td>
                <td><img src="$statusOption.getProperty('iconURL')"></td>
                #if($initialStatus.equals($statusId))
                    <td><input type="checkbox" disabled="true"
checked="checked"></td>
                #else
                    <td><input type="checkbox" disabled="true"></td>
                #end
                <td>$!statusOption.getProperty("description")</td>
            </tr>
        #end
        </table>
    #end

    ## --- 3. Generate Data for Transition Matrix (if selected) ---
    #if($SelectedDisplayDetails.contains("workflowTransitionList"))
        <h3 id="${wiTypeId}${uc}workflowTransitionList"> Workflow Transition
Matrix</h3>
        <table class="polarion-rpw-table-content" >
        <thead>
            <tr class="polarion-rpw-table-header-row" >
                <th style="border-right: 1px solid #cdd0d4"> </th>
                ## Create a header column for each status
                #foreach( $statusOption in $statusOptions )
                    <th style="border-left: 1px solid
#f0f0f0;"><center><b>$statusOption.getName()</b></center></th>
                #end
            </tr>
        </thead>
            ## Create a header row for each status
            #foreach( $colStatus in $statusOptions )
                #set($cName = $colStatus.getId())
                <tr class="polarion-rpw-table-content-row" >
                    <th style="border-right: 1px solid #cdd0d4;border-bottom:
1px solid #f0f0f0;">$colStatus.getName()</th>
                    ## Now iterate through all columns for this row
                    #foreach( $rowStatus in $statusOptions )
                        #set($rId = $rowStatus.getId())
                        #set($tranKey = "${cName}-${rId}")
                        #set($transitionId =
$stateTransitionMap.get($tranKey))
```

```
                        <td style="border-left: 1px solid
#f0f0f0;"><center>$!transitionId</center></td>
                    #end
                </tr>
            #end
        </table>
    #end
#end
```

Listing A-2: Macro for Link Role Graph Generation

This macro, `#GetlinkRoleDetails`, is responsible for building the link role graph. It iterates through *all* Work Item types in the project to check the valid link roles to and from the currently selected type. It then builds the JSON data structures for the nodes (Work Item types) and edges (link roles) to be rendered by `vis.js`.

Code snippet

```
##
================================================================================
============
## Macro: GetlinkRoleDetails
## Description: Fetches all incoming and outgoing link roles for a specific
##              Work Item type and builds the JSON graph data.
##
================================================================================
=============
#macro ( GetlinkRoleDetails $subType $allWorkItemTypes $linkRoleGrpahId
$prototypeModel)

    ## Get the master list of all link role enumerations in the project
    #set($linkRoleEnum = $currentProject.getWorkItemLinkRoleEnum() )
    #set($linkRoleOptions =
$linkRoleEnum.getAvailableOptions($linkRoleEnum.getControlKey())  )

    #set($nodesList = [] ) ## Holds the Work Item types
    #set($edgesList = [] ) ## Holds the links between them

    ## Add the "main" (selected) type to the node list first
    #foreach($wiType in $allWorkItemTypes )
        #if($subType.equals($wiType.id) )
            #if(!$nodesList.contains($wiType))
                #set($void = $nodesList.add($wiType))
            #end
        #end
    #end

    ## Iterate over every OTHER Work Item type to check for links
    #foreach($wiType in $allWorkItemTypes )
        #set($labelTags = "")     ## To store outgoing links (e.g.,
"verifies")
        #set($oppLabelTags = "")  ## To store incoming links (e.g., "is
verified by")
```

```
        ## Iterate over every possible link role (e.g., "verifies",
"duplicates", etc.)
        #foreach($lrOpt in $linkRoleOptions )
            #set($name = "")
            #set($oppName = "")

            ## Check for an OUTGOING link (from our type TO the other type)
            #if($lrOpt.isAllowed($subType, $wiType.id ) )
                #set($name = $lrOpt.name)
                #set($oppName = $lrOpt.oppositeName)
                #if(!$nodesList.contains($wiType))
                    #set($void = $nodesList.add($wiType)) ## Add target type
to graph
                #end
            ## Check for an INCOMING link (from the other type TO our type)
            #elseif( $lrOpt.isAllowed( $wiType.id, $subType ) )
                #set($name = $lrOpt.oppositeName)
                #set($oppName = $lrOpt.name)
                #if(!$nodesList.contains($wiType))
                    #set($void = $nodesList.add($wiType)) ## Add target type
to graph
                #end
            #end

            ## Consolidate multiple link types onto one arrow
            #if(!$name.equals(""))
                #set($labelTags = $labelTags.concat(${name}).concat(","))
                #set($oppLabelTags =
$oppLabelTags.concat(${oppName}).concat(","))
            #end
        #end

        ## Clean up trailing comma
        #if($labelTags.length() > 0)
            #set ($stringLength = $labelTags.length() - 1)
            #if($labelTags.substring($stringLength).equals(","))
                #set ($labelTags = $labelTags.substring(0,$stringLength))
            #end
        #end
        #if($oppLabelTags.length() > 0)
            #set ($stringLength = $oppLabelTags.length() - 1)
            #if($oppLabelTags.substring($stringLength).equals(","))
                #set ($oppLabelTags =
$oppLabelTags.substring(0,$stringLength))
            #end
        #end

        ## Create the JSON edge objects for vis.js
        #if($labelTags.length() > 0)
            #set($dirEdge = "{${q}from${q}:${q}${subType}${q},
${q}to${q}:${q}$wiType.id${q}, ${q}label${q}:${q}${labelTags}${q}}")
            #set($void = $edgesList.add($dirEdge))
        #end
        #if($oppLabelTags.length() > 0)
            #set($oppEdge = "{${q}from${q}:${q}$wiType.id${q},
${q}to${q}:${q}${subType}${q}, ${q}label${q}:${q}${oppLabelTags}${q}}")
```

```
            #set($void = $edgesList.add($oppEdge))
        #end
    #end

    ## --- Build final JSON for Nodes and Edges ---
    #set($linknodes = "[")
    #set($linkedges = "[")
    #set($firstNode = true)
    #set($firstEdge = true)

    #foreach($node in $nodesList )
        #set($nodeName = $node.name)
        #set($nodeStr =
"{${q}id${q}:${q}$node.id${q},${q}label${q}:${q}$nodeName${q}}")
        #if($firstNode) #set($firstNode = false) #else #set($linknodes =
"${linknodes},") #end
        #set($linknodes = "${linknodes} ${nodeStr}")
    #end

    #foreach($edgeStr in $edgesList )
        #if($firstEdge) #set($firstEdge = false) #else #set($linkedges =
"${linkedges},") #end
        #set($linkedges = "${linkedges} ${edgeStr}")
    #end

    #set($linknodes = "${linknodes}]" )
    #set($linkedges = "${linkedges}]" )

    ## Create the HTML container for the graph
    <div id='$linkRoleGrpahId' class='graphh'"></div>

    ## Inject the JSON data into <script> tags for the client-side
    <script>
        var _nodes$linkRoleGrpahId = '${linknodes}';
        var _edges$linkRoleGrpahId = '${linkedges}';
    </script>
#end
```

B. Detailed API Interactions

This appendix documents the specific HTTP requests, headers, and JSON payloads used by the **DataFetcherService** to facilitate communication between the React widget and the Polarion ALM backend. It serves as a technical reference for the "Direct-to-REST" architecture described in **Chapter 6**.

B.1 Authentication and Request Headers

As detailed in Section **6.2.3**, the widget utilizes a session-based token to authenticate without prompting the user for credentials. Every HTTP request initiated by the React application includes the following standard headers:

`Accept`: `application/json` (Ensures the response is parsable data).

`X-Polarion-Rest-Token`: `{sessionToken}` (The token injected by Velocity during initialization).

Request Header Example:

JSON

```
{
  "Accept": "application/json",
  "Content-Type": "application/json",
  "X-Polarion-Rest-Token": "12345-abcde-67890-fghij"
}
```

B.2 Fetching Work Items (Core Data)

To generate the "Workflow Overview" and "Link Role" graphs, the widget must first retrieve the relevant Work Items. To optimize performance, the request uses the `fields` parameter to request only necessary metadata, reducing payload size.

Endpoint: `/polarion/rest/v1/projects/{projectId}/workitems`

Method: `GET`

Query Parameters:

`query`: `type:{workItemType} AND status:{status}` (Dynamic Lucene query).

`fields`: `id,title,status,type,linkedWorkItems`

`limit`: `100` (Used with pagination).

Sample Response Payload (JSON):

JSON

```
{
  "data": [
    {
      "type": "workitems",
      "id": "VALEO-1024",
      "attributes": {
        "id": "VALEO-1024",
        "title": "Implement CAN Bus Interface",
        "status": "inprogress",
        "type": "softwareTask"
      },
      "relationships": {
        "linkedWorkItems": {
          "data": [
```

```
            {
              "type": "workitems",
              "id": "VALEO-890",
              "meta": {
                "role": "implements"
              }
            }
          ]
        }
      },
      "links": {
        "self":
"https://polarion.valeo.com/polarion/rest/v1/projects/PEM/workitems/VALEO-
1024"
      }
    }
  ],
  "meta": {
    "totalCount": 45
  }
}
```


B.3 Retrieving Workflow Configurations

While static workflow definitions are often fetched via the Java API (see Appendix A), dynamic state transitions for specific items are validated via the REST API to ensure the "Next Steps" shown to the user are valid.

Endpoint:

`/polarion/rest/v1/projects/{projectId}/workitems/{workItemId}/actions`

Method: `GET`

Sample Response Payload (Available Transitions):

JSON


```
{
  "data": [
    {
      "type": "actions",
      "id": "action_mark_verified",
      "attributes": {
        "nativeActionId": "markVerified",
        "nativeActionName": "Mark as Verified",
        "targetStatus": "verified"
      }
    },
    {
      "type": "actions",
      "id": "action_reject",
      "attributes": {
        "nativeActionId": "reject",
```

```
        "nativeActionName": "Reject Task",
        "targetStatus": "rejected"
      }
    }
  ]
}
```

B.4 Fetching Enumerations (Status & Types)

To populate dropdown menus (e.g., the "Work Item Type" selector in Figure C-1) without hardcoding values, the widget dynamically queries the project's enumeration configurations.

Endpoint: `/polarion/rest/v1/projects/{projectId}/enumerations/status`

Method: `GET`

Sample Response Payload:

JSON

```
{
  "data": [
    {
      "type": "enumerations",
      "id": "open",
      "attributes": {
        "name": "Open",
        "color": "#ff0000",
        "iconURL": "/polarion/icons/status/open.png"
      }
    },
    {
      "type": "enumerations",
      "id": "inprogress",
      "attributes": {
        "name": "In Progress",
        "color": "#00ff00",
        "iconURL": "/polarion/icons/status/inprogress.png"
      }
    }
  ]
}
```

B.5 Error Handling Response Structure

The React widget is designed to gracefully handle API failures. The State Machine (Figure 6.4) relies on standard HTTP status codes to trigger the "Error State."

Common Error Response:

JSON

```
{
  "errors": [
    {
      "status": "403",
      "title": "Access Denied",
      "detail": "The current user does not have permission to read Work Items
in project 'MasterThesis_Sandbox'."
    }
  ]
}
```

- **401 Unauthorized:** Triggered if `X-Polarion-Rest-Token` is invalid or expired (prompts user to refresh).
- **403 Forbidden:** Triggered if the user lacks "READ" permissions for the specific Work Item Type.
- **404 Not Found:** Triggered if the `projectId` injected by Velocity does not exist.

C. Sample Documentation Output

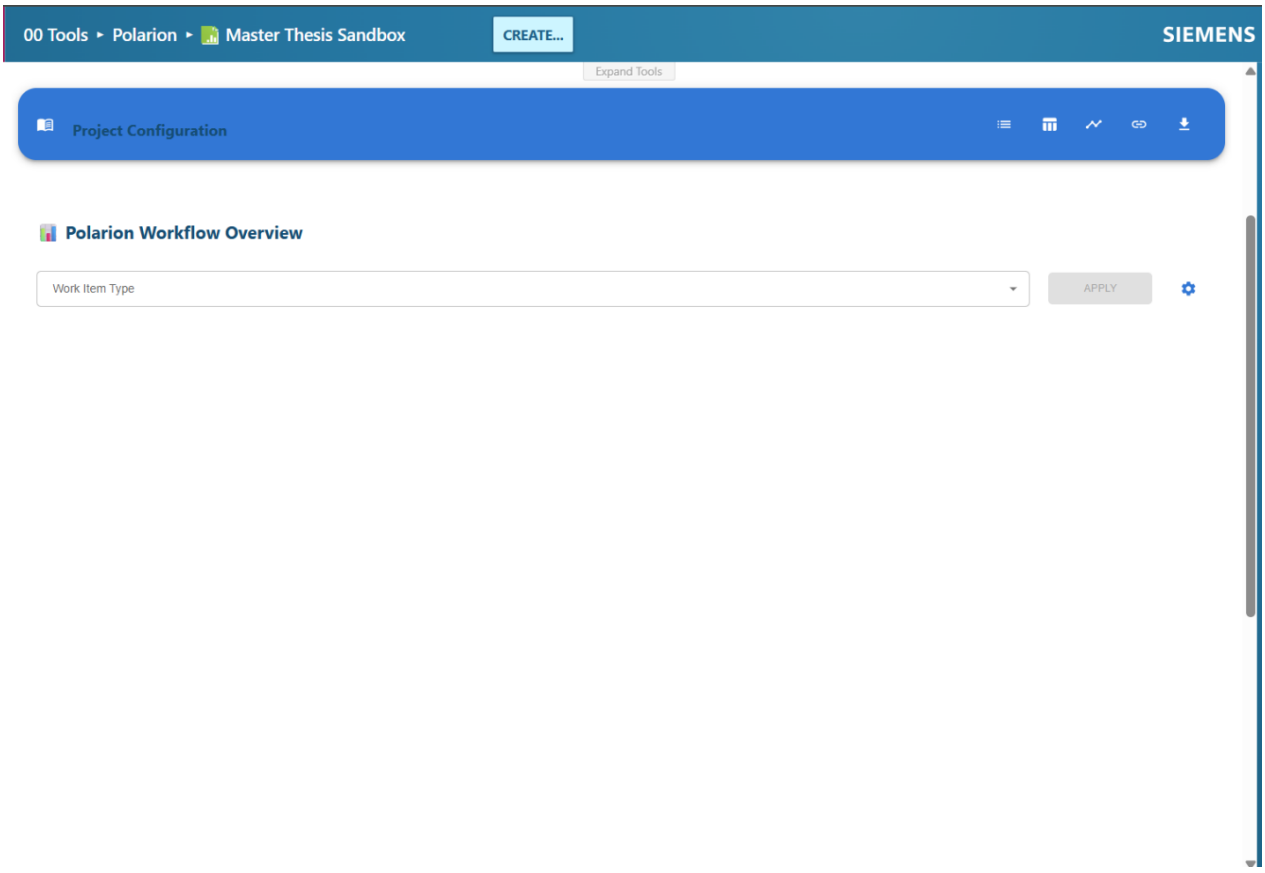


Figure C-1: Widget Initial State *The widget as it first appears on a Polarion Rich Page, with the "Work Item Type" selection dropdown empty.*

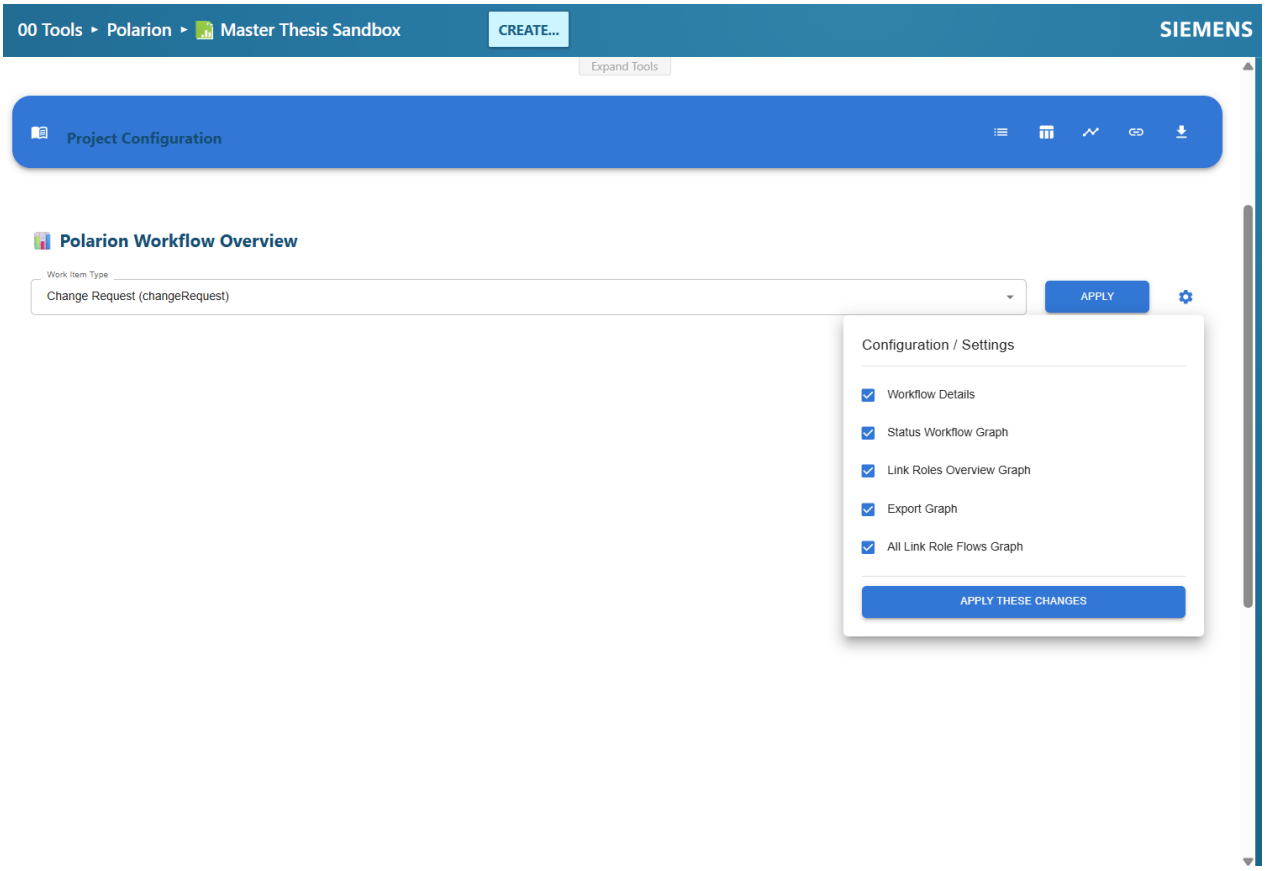


Figure C-2: Widget Configuration Menu *The widget's settings panel, allowing the user to select which reports to generate, such as "Workflow Details," "Status Workflow Graph," and "Link Roles Overview Graph."*

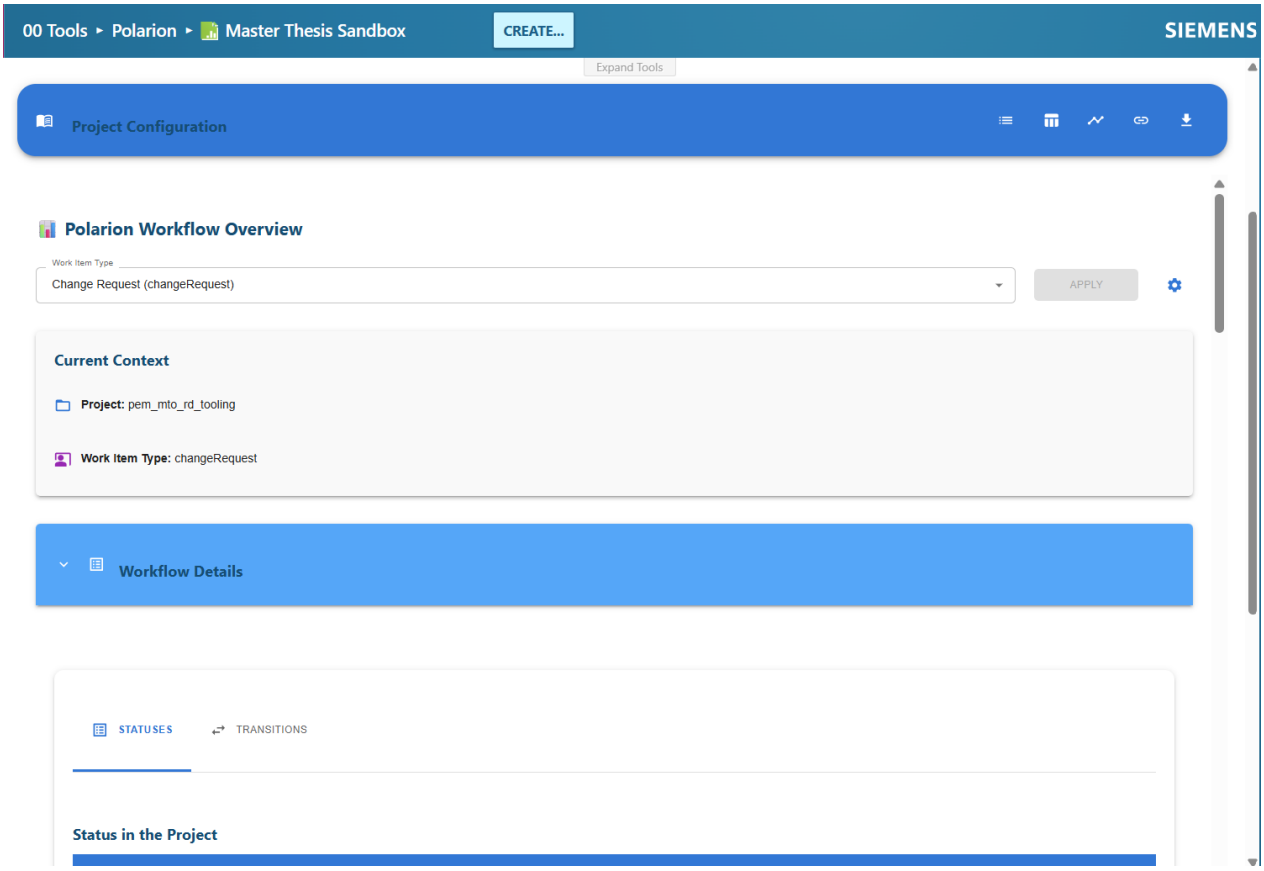


Figure C-3: Context After Work Item Selection *After selecting a "Work Item Type" (in this case, "ChangeRequest"), the widget displays the current project context and the collapsible report sections.*

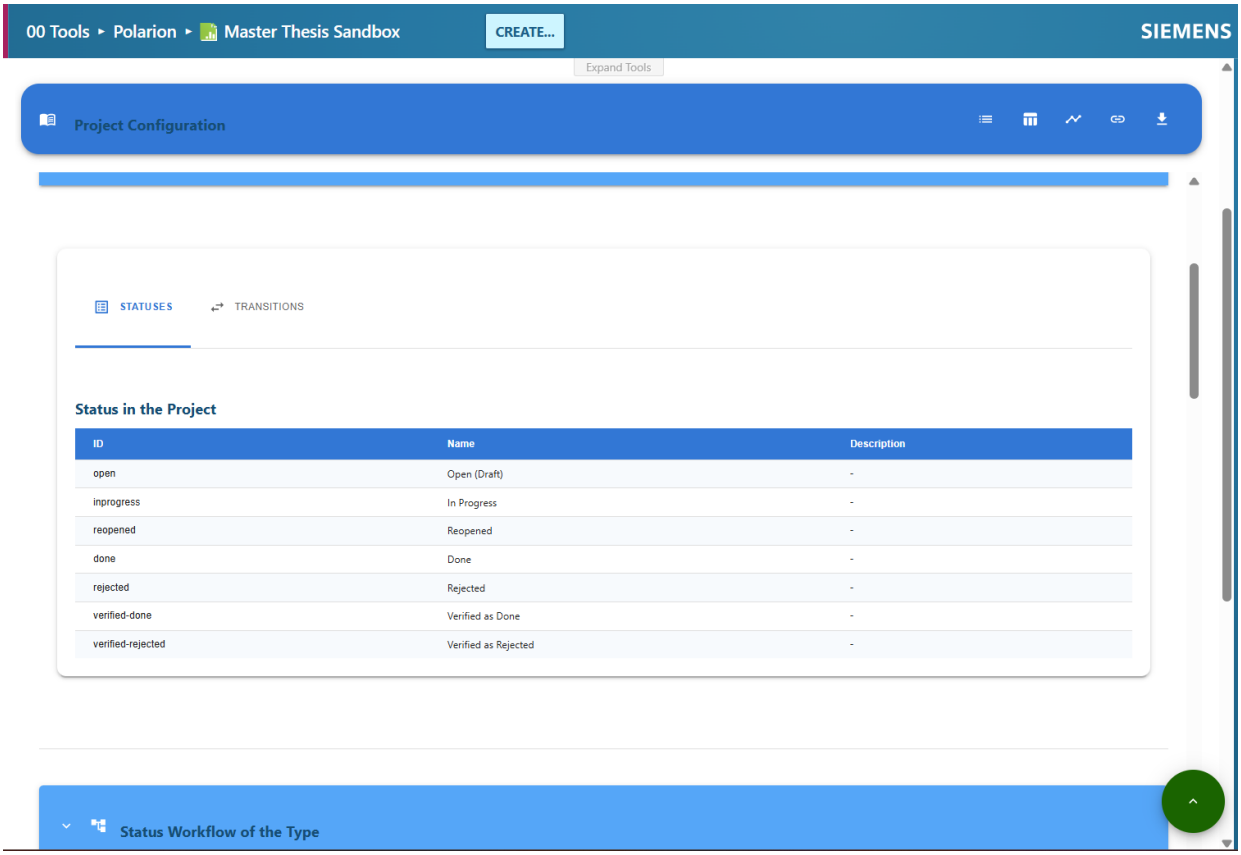


Figure C-4: Tabular Report - Statuses *The "Workflow Details" report, showing the "Statuses" tab. This table lists all configured statuses for the selected "ChangeRequest" type, such as open, inprogress, and reopened.*

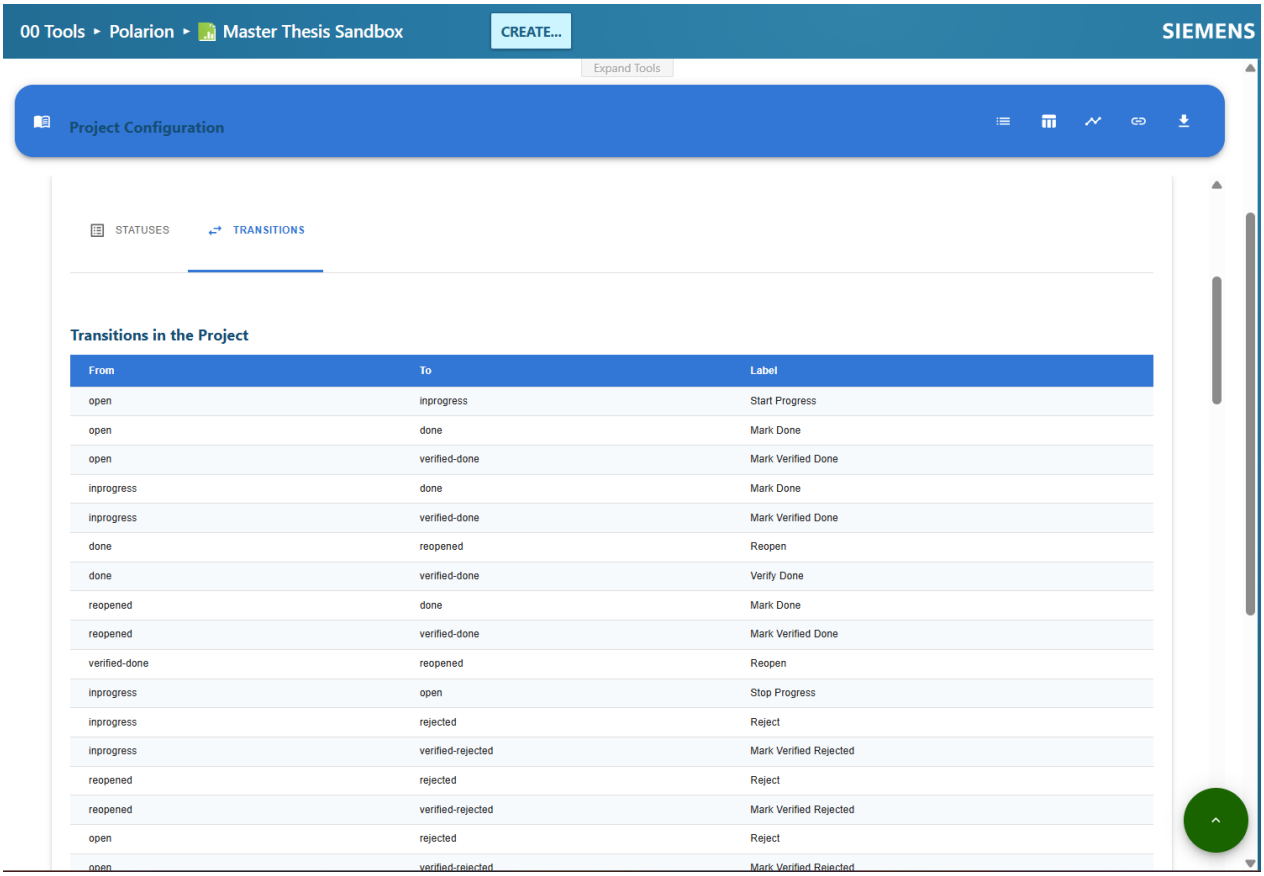


Figure C-5: Tabular Report - Transitions *The "Transitions" tab for the same Work Item. This provides a clear, tabular view of the workflow logic, showing the "From" state, "To" state, and the "Label" for the action.*

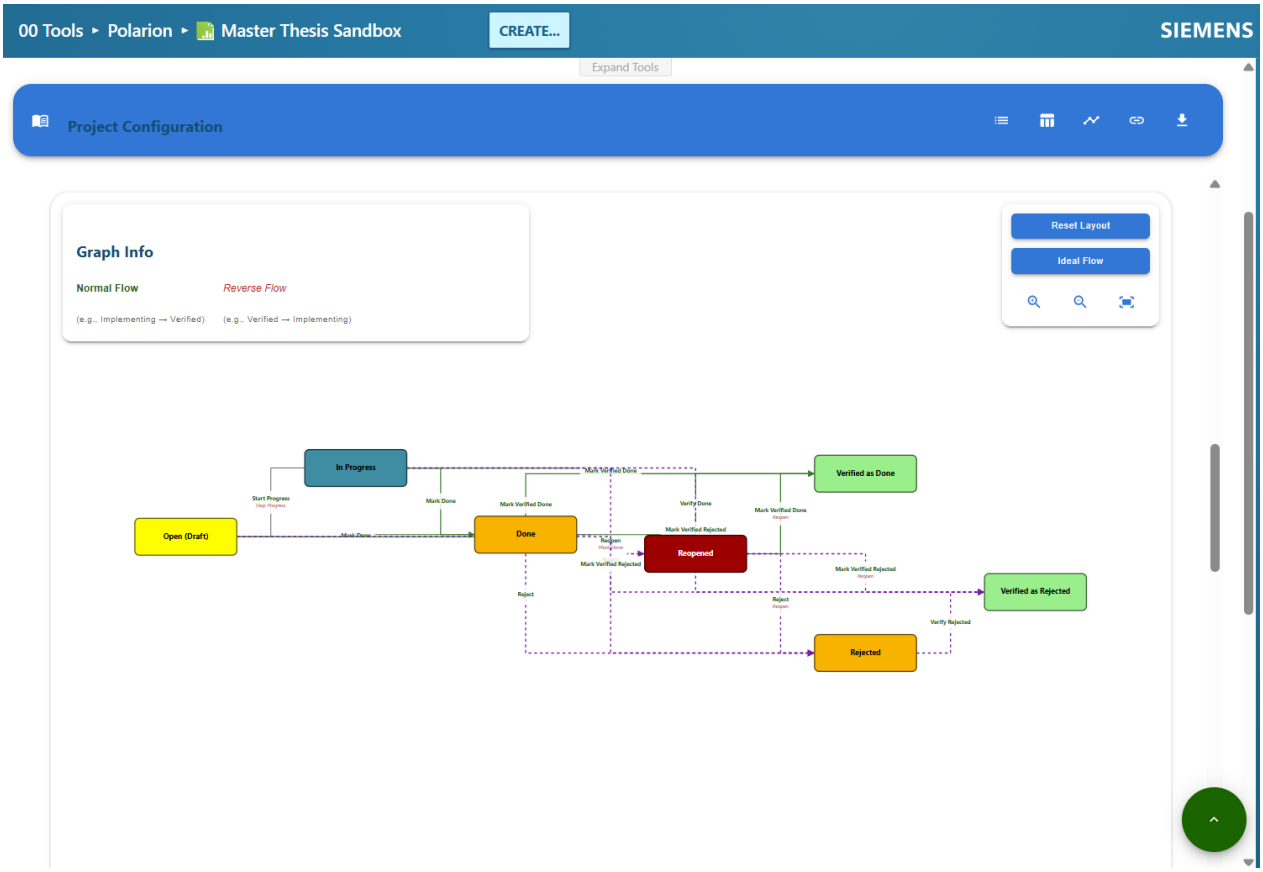


Figure C-6: Generated Status Workflow Graph *A key feature of the widget: the dynamic visualization of the "ChangeRequest" workflow. This graph is generated using VisJS and clearly illustrates the flow from Open (Draft) to Verified as Done or Verified as Rejected.*

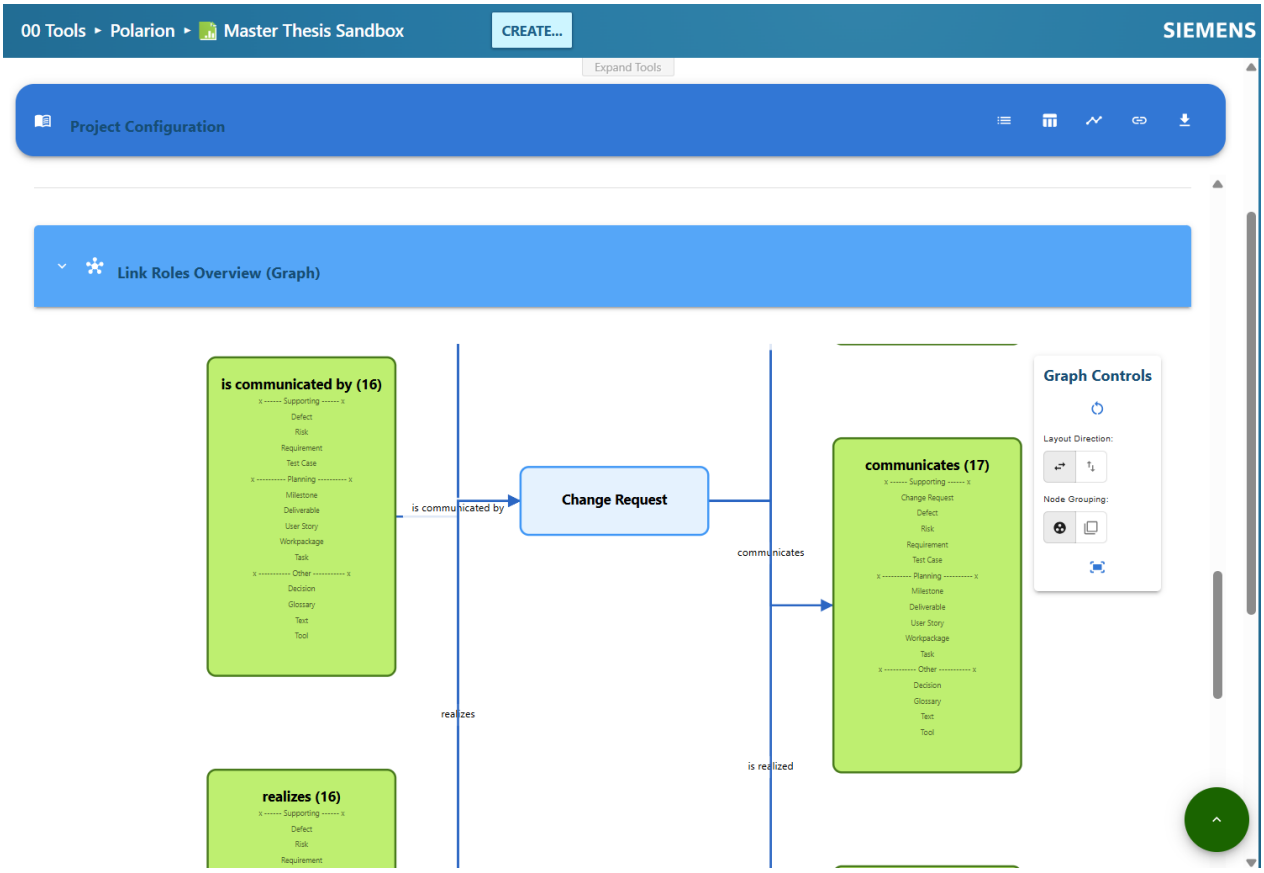


Figure C-7: Generated Link Roles Overview Graph *The "Link Roles Overview (Graph)" for the "Change Request" type. This visualization demonstrates how the selected Work Item interacts with other project artifacts, such as "is communicated by" and "realizes."*

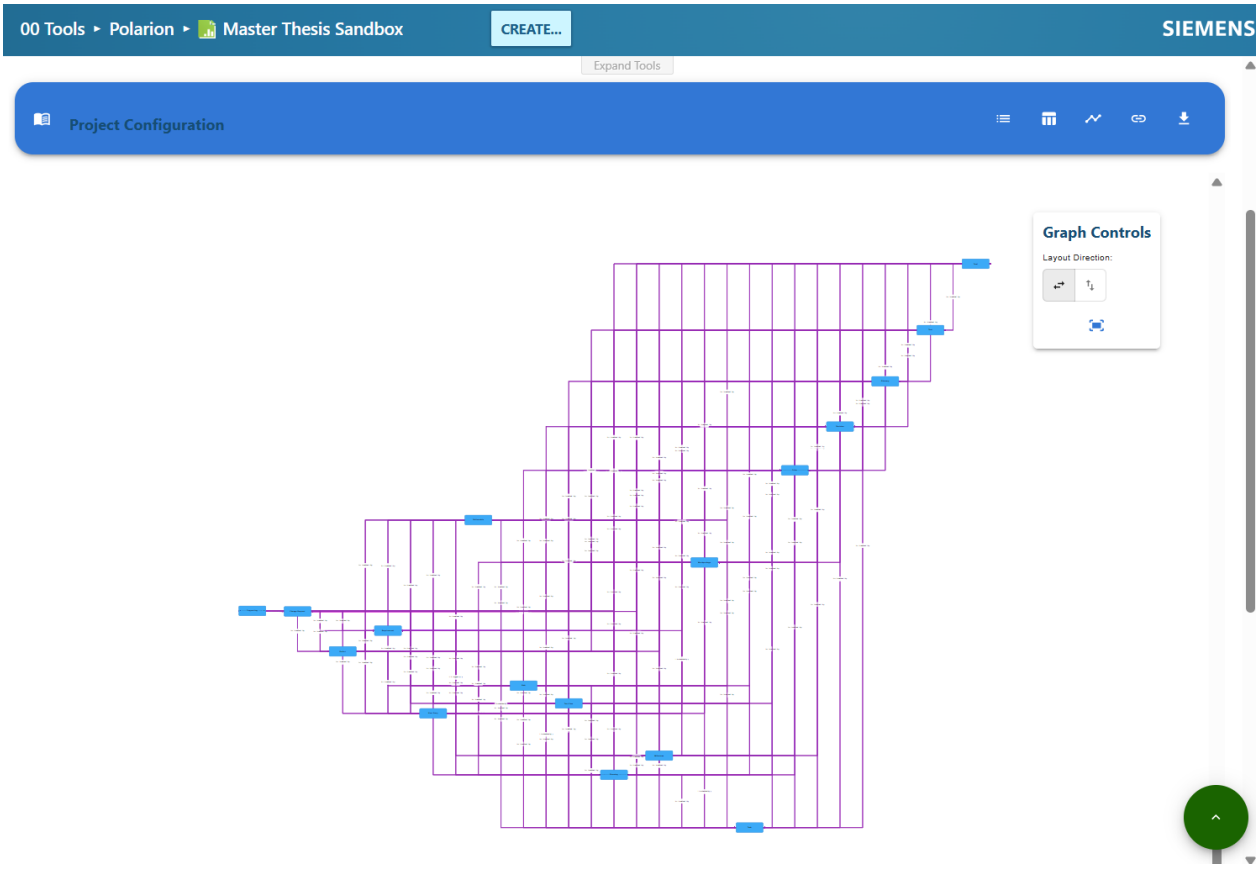


Figure C-8: Generated "All Link Role Flows Graph" *An advanced reporting feature showing all possible link role relationships within the entire project. This complex graph helps in advanced configuration maintenance and process analysis.*

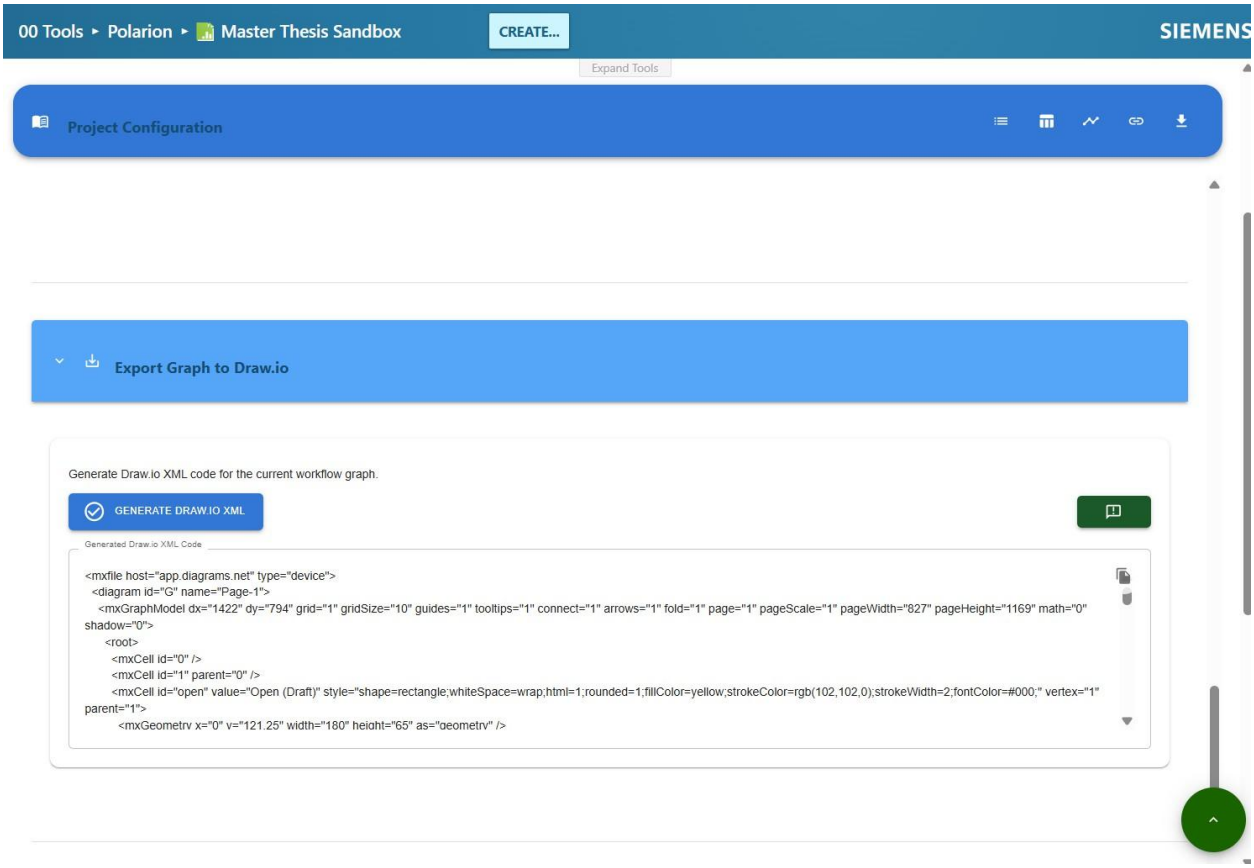


Figure C-9: Export to Draw.io Feature *An additional utility feature that generates the Draw.io XML code for the current workflow graph, allowing engineers to export the visualization for use in other external documentation or diagrams.*

D. Glossary

1. ALM **(Application Lifecycle Management):** A comprehensive system for managing the entire lifecycle of a software product from its conception, through design, development, and testing, to its release and maintenance. The platform used in this thesis is Siemens Polarion.
2. **API (Application Programming Interface):** A set of definitions, protocols, and tools that allows different software applications to communicate with each other. This research utilizes Polarion's Java API for server-side data access and its REST API for client-side interactions.
3. **ASPICE (Automotive SPICE):** A process assessment model for the automotive industry, used to evaluate and improve the maturity of software development processes. Adherence is critical for automotive suppliers.

4. **Component-Based Architecture:** A software design approach where an application is built from a set of modular, reusable, and independent components. This is a core principle of the React JS framework.
5. **Data Model:** The abstract structure of data in a system, defining its elements (like Work Items) and the relationships between them (like Link Roles). This thesis generates documentation by reading Polarion's live data model.
6. **Documentation Drift:** A common software engineering challenge where official documentation gradually becomes outdated, inconsistent, and no longer accurately reflects the current state of the implemented software or configuration.
7. **Functional Requirement (FR):** A requirement that defines a specific behavior, function, or "what" the system is supposed to do (e.g., "The system shall extract data models").
8. **HIP (Hybrid Integration Platform):** An architectural framework that enables the integration of on-premise systems (like a local Polarion server) with modern cloud or web-based applications (like a React JS widget).
9. **ISO 26262:** An international standard for the functional safety of electrical and electronic (E/E) systems in road vehicles. It mandates strict documentation and traceability throughout the development process.
10. **Link Role:** A specific term within Polarion that defines the nature of the relationship between two Work Items. Examples include "verifies," "realizes," "is related to," or "is communicated by."
11. **Non-Functional Requirement (NFR):** A requirement that specifies criteria for the system's operation rather than its specific functions. This includes attributes like performance, security, usability, and maintainability.
12. **Polarion:** An enterprise-grade ALM platform by Siemens. It serves as the central "single source of truth" for all project artifacts (requirements, tests, etc.) and is the host environment for the widget developed in this thesis.
13. **React JS:** A popular open-source JavaScript library for building user interfaces, based on a component-based architecture. It is used in this thesis to create the dynamic, client-side frontend of the documentation widget.

14. **React Hooks:** Functions (such as `useState` and `useEffect`) that allow developers to "hook into" React's state and lifecycle features from functional components, simplifying state management and side effects like data fetching.
15. **SDK (Software Development Kit):** A collection of software development tools in one installable package, provided by a platform vendor. This thesis uses Polarion's SDK, specifically its Java API, to access server-side data and services.
16. **State Management:** The practice of managing the data that changes over time in an application. In this thesis, this refers to React's use of hooks like `useState` to manage data fetched from Polarion.
17. **Traceability:** The ability to follow the life of an artifact (like a requirement) throughout the development lifecycle, from its origin to its implementation and verification. This is a core tenet of both ISO 26262 and ASPICE.
18. **UI/UX (User Interface / User Experience):** UI refers to the visual layout and controls a user interacts with. UX refers to the user's overall feeling and satisfaction while interacting with the application.
19. **Velocity (Template Engine):** An open-source, Java-based template engine used by Polarion to render web pages (VTL). In this architecture, it acts as the "bridge" to inject server-side Java data into the client-side React application.
20. **Virtual DOM:** A core concept in React where a lightweight, in-memory representation (a "virtual" Document Object Model) is maintained. This allows React to efficiently calculate and apply updates to the browser, leading to high performance.
21. **Work Item:** The fundamental unit of data in Polarion, representing any artifact in the lifecycle, such as a Requirement, Test Case, Defect, or Change Request.
22. **Workflow:** The defined sequence of states (e.g., "Open," "In Progress," "Done") and transitions (e.g., "Start Progress") that a Work Item must follow during its lifecycle.

List of Tables

List of Figures