

POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING



MASTER's Degree Thesis

Modernising Legacy Eclipse RCP Tooling: A Migration Framework for Web-Based IDE Environments

Supervisors

Prof. Marco TORCHIANO

Ing. Enrico BRAGANTE

Candidate

Andrea GALELLA

March 2026

Modernising Legacy Eclipse RCP Tooling: A Migration Framework for Web-Based IDE Environments

Andrea Galella

Abstract

In recent years, the information technology sector has witnessed a persistent shift towards web-based and cloud environments. The need for flexibility brings tools development increasingly towards the web, by the use of stand-alone web applications capable of integrating in modern web-stack IDEs like Visual Studio Code and Eclipse Theia. As a native-focused field, embedded system development strictly relied on desktop IDEs and native-OS widgets due to hardware integration needs and real-time constraints. The thesis focuses on highlighting the reasons, problems and methodologies of migrating Eclipse RCP applications to the web. Three main migration strategies are implemented and integrated within a web-based IDE environment, showcasing architectural migration and emerging technical issues. The Eclipse RAP pathway aims to be a conservative and minimal-change approach; the React.js pathway works as the web-native re-architecture; and lastly the WebAssembly with Slint pathway is the most modern and heavy-resource-orientated approach. Two practical application's use cases for the field of microcontroller tooling in the automotive industry are finally used to highlight the results and assess a modernisation framework tailored to diverse requirements.

Table of Contents

1	Introduction	1
1.1	Background and Contextual Foundation	1
1.1.1	Eclipse Rich Client Platform (RCP) and Domain-Specific Tooling	1
1.1.2	The Foundational Limits of the Thick Client Model	2
1.2	Motivation for Modernisation: Economic and Strategic Drivers . . .	3
1.2.1	The Critical Shift in User and Business Expectations	3
1.2.2	Total Cost of Ownership (TCO) Escalation and Talent Scarcity	3
1.2.3	The Architectural Imperative of Cloud-Native Adaptation . .	4
1.3	Issues and Architectural Challenges of Eclipse RCP Migration . . .	5
1.3.1	Technical Debt Manifestations Specific to RCP	5
1.3.2	The Migration Paradox: Rewrite vs Reuse	6
1.4	Emerging Eclipse Migration Targets for Domain-Specific Tooling . .	7
1.4.1	Visual Studio Code: The Dominant Extension Ecosystem . .	7
1.4.2	Eclipse Theia: The Cloud-Native Workbench Successor . . .	7
1.5	Modernisation Methodologies	9
1.5.1	Strategic Pathways and Application Modernisation Models .	9
1.5.2	Incremental and Transitional Strategies	9
1.5.3	Eclipse RAP	9
1.5.4	Eclipse GLSP: The Future of Diagram Editors	10
1.5.5	Equo SWT Evolve: The "Lift-and-Shift" Bridge	10
1.5.6	Web Technologies	10
1.6	Case study: Microcontroller Tooling in the Automotive industry . .	13
1.6.1	Infineon Technologies and Aurix Configuration Studio (ACS)	13
1.7	Structure of the Thesis and Original Contribution	15
1.7.1	Research Questions and Hypotheses	15
1.7.2	Overview of the Research Methodology	15
1.7.3	Original Contribution and Thesis Structure	15
2	Literature Review and Architectural Analysis	17
2.1	Comparative Analysis of Platform Architectures	17
2.1.1	Classic Eclipse RCP: The OSGi Monolith	17
2.1.2	Eclipse RAP: The Headless Workbench	18
2.1.2.1	RAP Threads	20
2.1.3	Visual Studio Code: Multi-Process Sandbox	20

2.1.3.1	Extension manifest	22
2.1.3.2	Views	22
2.1.3.3	Webview API	22
2.1.3.4	Message Passing with the JSON-RPC protocol . . .	23
2.1.4	Eclipse Theia: The Hybrid Web Platform	24
2.1.4.1	Extension Model	24
2.1.5	Special-Purpose Architectures (Equo SWT Evolve, Eclipse GLSP)	25
2.2	Web Frontend Frameworks	27
2.2.1	Java on the Frontend: Architectural Friction and Performance Overheads	27
2.2.1.1	Server-driven UI	27
2.2.1.2	Transpiling	28
2.2.2	JavaScript and React: The Industry Standard and its Constraints	28
2.2.3	Rust + WebAssembly (Wasm): The Native-Speed Runtime .	29
2.2.3.1	The Yew Framework: Component-Based Rust for the Frontend	30
2.2.3.2	Slint: Compiled user interface for High-Performance Tooling	30
3	Methodology and Design	31
3.1	Scientific Foundation and Research Philosophy	31
3.2	Baseline Assessment: Technical Debt and Code Recyclability Analysis	32
3.2.1	Quantitative Models for Technical Debt	32
3.2.2	Java Code Recyclability Assessment	33
3.3	Taxonomy of Migration Strategies: Preservation vs. Web-Native Approaches	33
3.3.1	Exclusion of Partial Modernisation Approaches	34
3.4	Case Study Selection and Implementation Scope	34
3.4.1	Case Study 1: Standard Form-Based Application	34
3.4.2	Case Study 2: Graphics-Intensive Application	35
3.4.3	Implementation Process	36
3.5	Architectural Metrics for Cross-Platform Comparison	36
3.5.1	Primary Evaluation Measures: Maintainability and Code Quality	36
3.5.2	Secondary Measurement Metrics: Performance and End-User Interaction	37
3.5.3	Cloud-Native Readiness Assessment	37
3.5.4	Comparative Study: Tools and Parameters	38
3.6	Strategic Decision Framework	38
3.6.1	RAP Evaluation Criteria	38
3.6.2	React vs. Wasm Selection Framework	39
3.7	Research Execution Summary	39
3.7.1	Phase 1: Universal Maintainability Assessment	39

3.7.2	Phase 2: Standard Application Comparison (React vs. Wasm)	39
3.7.3	Phase 3: Validation with Graphics-Intensive Applications . .	40
4	Implementation of Modernisation Pathways	41
4.1	Custom SWT Widgets	41
4.2	Implementation Strategy 1: Eclipse RAP	43
4.2.1	Architectural Mapping	43
4.2.2	Entry Point Configuration and Lifecycle	43
4.2.3	The Headless State API: REST Controller Integration	44
4.2.4	Technical Challenges and Solutions	44
4.2.4.1	Key constants	44
4.2.4.2	Network latency	45
4.2.4.3	Link handling	46
4.2.5	Thread Synchronisation	46
4.2.6	SRAPA: Spring RAP Automation	48
4.3	Implementation Strategy 2: React	49
4.3.1	Architectural Mapping: The "Workbench" to "DOM" Translation	49
4.3.2	Workbench Replication and Layout Management	50
4.3.3	Enterprise-Scale State Management	52
4.3.3.1	Context Optimisation: The Selector Pattern	53
4.3.4	Remote Data Synchronisation	54
4.3.5	Bidirectional Integration: The Headless State API	54
4.3.6	Technical Challenges and Solutions	54
4.3.6.1	The "Blocking Dialogue" Problem	54
4.3.6.2	The Broadcast Problem	56
4.4	Implementation Strategy 3: WebAssembly with Slint	57
4.4.1	Architectural Mapping: From Widget-Based to Canvas-Based Rendering	57
4.4.2	Declarative UI Definition	58
4.4.3	State Management: Rust Ownership vs. Java GC	60
4.4.3.1	Memory Management Patterns	60
4.4.4	The Wasm-JS Bridge for External Connectivity	61
4.4.5	Build Pipeline and Deployment	64
4.4.6	Technical Challenges and Solutions	64
4.4.6.1	Institutional Knowledge Attrition and Documenta- tion Deficiencies	64
4.4.6.2	Feature Parity vs. The Innovation Tax	65
4.4.6.3	Wasm Ecosystem Limitations	66
5	Integration with IDE Ecosystems	67
5.1	Integration based on WebView	68
5.1.1	URI Resolution and Local Resource Access	68
5.1.2	Dynamic HTML Content Generation and Security	68

5.1.3	The Communication Protocol: Type-Safe JSON-RPC	69
5.1.3.1	Mechanism of the RPC Lifecycle	70
5.1.4	Workspace and Filesystem Interoperability	70
5.1.5	The WebAssembly Bootstrapping Process	71
5.1.5.1	The Communication Protocol: Async RPC over FFI	72
5.1.5.2	The Virtual Filesystem (VFS) Layer	73
5.1.5.3	Technical Challenge: Serialisation and the "Wasm Boundary"	73
5.2	Integration of native Theia extensions	74
5.2.1	Architectural Foundation: Inversify.js Dependency Injection .	75
5.2.2	Widget-Based UI Integration	75
5.2.2.1	Example: Migrating an Eclipse RCP View to a Theia Widget	75
5.2.2.2	Key Architectural Differences from the WebView Ap- proach	77
5.2.3	Dependency Injection Module Configuration	77
5.2.4	Integration of Backend Services	78
5.2.5	Filesystem Integration: Direct API Access	79
5.3	The Exception: Eclipse RAP	81
5.3.1	Fundamental Architectural Divergence	82
5.3.2	Complexity of RAP-IDE Integration	82
5.3.3	Integration Strategy: Headless Java Sidecar	83
6	Empirical Analysis and Comparative Evaluation	87
6.1	Subject Applications: Architectural Profile and Modernisation Goals	87
6.1.1	Case Study 1: Clock Configuration Application (Standard Form-Based Tool)	88
6.1.2	Case Study 2: Automotive Bus Monitor (Graphics-Intensive Tool)	88
6.1.3	Planned Application Selection Rationale	90
6.2	Phase 1: Universal Maintainability Assessment	91
6.2.1	Eclipse RAP Implementation: Maintainability Analysis . . .	91
6.2.1.1	Code Reuse Achievement	91
6.2.1.2	Technical Debt Metrics	91
6.2.2	React Implementation: Maintainability Analysis	92
6.2.2.1	Code Reuse Achievement	92
6.2.2.2	Technical Debt Metrics	92
6.2.3	Slint/WebAssembly Implementation: Maintainability Analysis	93
6.2.3.1	Code Reuse Assessment	93
6.2.3.2	Technical Debt Metrics	93
6.2.4	Phase 1 Summary: Production-Readiness Validation	94
6.3	Phase 2: Architectural Evaluation and RAP Exclusion Analysis . . .	95
6.3.1	Cloud-Native Compatibility Assessment	95

6.3.1.1	Stateful Server Architecture and Constraints on Horizontal Scaling	95
6.3.1.2	Network Latency and Impact on User Experience	95
6.3.2	Development Workflow and Integration of Ecosystems	96
6.3.2.1	IDE Integration and Modern Tooling Compatibility	96
6.3.2.2	Third-Party Library Ecosystem	97
6.3.3	Strategic Exclusion Decision: RAP as a Transitional Solution	97
6.4	Phase 3: Focused Comparison of Web-Native Pathways	98
6.4.1	Standard Business Application: Clock Configuration Tool	98
6.4.1.1	Performance Baseline Comparison	98
6.4.2	Graphics-Intensive Application: Automotive Bus Editor	99
6.4.2.1	Rendering Performance Under Load	99
6.4.2.2	Memory Performance and Predictability	100
6.5	Cross-Cutting Analysis: Strategic Decision Framework	101
6.5.1	Comparative Summary: All Pathways	101
6.5.2	Key Empirical Findings	101
6.5.3	Strategic Recommendations	102
6.6	Chapter Summary: Empirical Testing of Strategic Framework	103
7	Conclusion and Prospective Work	105
7.1	Research Results	105
7.1.1	Strategic Decision Framework for Eclipse RCP Modernisation	105
7.1.2	Empirical Confirmation of Code Reuse Paradox	106
7.1.3	Performance Boundary Characterisation for Web-Native Architectures	106
7.1.4	Architectural Patterns for Cross-Platform IDE Integration	107
7.1.5	Automation Tooling: Spring RAP Automation (SRAPA)	108
7.1.6	Methodological Contribution: Application of Design Science Research	108
7.2	Answers to Research Questions	109
7.2.1	RQ1: How can legacy Eclipse RCP applications be effectively refactored to minimise technical debt while maximising business logic reuse?	109
7.2.2	RQ2: What are the quantifiable compromises between migration pathways prioritising Java code reuse versus those requiring UI rewrite for cloud-native optimisation?	110
7.2.3	RQ3: Which modern web technology stack is best suited for a full UI overhaul in embedded systems tooling?	111
7.2.4	RQ4: Can a strategic modernisation framework help enterprise architects determine the most suitable migration strategies for domain-specific tool ecosystems?	112
7.3	Limitations of the Study	115
7.3.1	Application Domain Specificity	115

7.3.2	Technology Maturity Evolution	115
7.3.3	Scalability Testing Constraints	116
7.3.4	Dependence on Organisational Context	117
7.4	Future Research Directions	118
7.4.1	Automated Migration Tooling	118
7.4.2	Polyglot Architecture Patterns for Hybrid Tool Suites	119
7.4.3	AI-Assisted Migration and Modernisation	119
7.4.4	Accessibility and inclusive design in Migrated Applications	120
7.5	Concluding Remarks	121
Bibliography		123

List of Figures

2.1	VS Code extension host architecture [43]	22
2.2	Theia extension types [47]	25
4.1	UML class diagram - custom package internal implementations . . .	42
4.2	UML class diagram - custom package external relationships	42
4.3	Character fields and corresponding keyCode	45
4.4	Code difference between SWT and RWT Mouse Wheel listener imple- mentation	46
4.5	Code difference between SWT and RWT link handling	46
4.6	Spring RAP Automation (SRAPA)	48
5.1	Tool application running in a VS Code WebView	71
5.2	Deployment diagram - WebView	73
5.3	Component diagram - WebView	74
5.4	Sequence diagram - WebView	74
5.5	Tool application running in a Theia extension	77
5.6	Deployment diagram - Theia extension	80
5.7	Component diagram - Theia extension	81
5.8	Sequence diagram - Theia extension	81
5.9	Deployment diagram - Eclipse RAP and VS Code	84
5.10	Component diagram - Eclipse RAP and VS Code	85
5.11	Sequence diagram - Eclipse RAP and VS Code	85
5.12	Comparison architecture diagrams	86
6.1	Clock Configuration Application - Legacy Eclipse RCP Implementation	89
6.2	Automotive Bus Editor	90

List of Tables

1.1	Comparative Analysis of TCO Drivers in Application Architectures .	4
1.2	Legacy Application Age and Modernisation Challenges [13]	5
1.3	Architectural Evaluation of Eclipse Migration Target Platforms . . .	12
2.1	RAP setups	19
3.1	Maintainability Rating and Technical Debt Ratio	32
3.2	Migration Technology Scope and Research Inclusion Rationale . . .	34
3.3	Comparative Study Tools and Parameters	38
3.4	Strategic Pathway Selection Criteria	39
3.5	Strategic Pathway Comparison Summary	40
4.1	Component Architectural Mapping with React	50
4.2	Component Architectural Mapping with Slint	58
5.1	Comparison of Integration Approaches	77
6.1	Complementary Validation Roles of Subject Applications	90
6.2	Eclipse RAP Implementation - Technical Debt Metrics (Clock Configuration App)	91
6.3	React Implementation - Technical Debt Metrics (Clock Configuration App)	92
6.4	Slint/WebAssembly Implementation - Technical Debt Metrics (Clock Configuration App)	94
6.5	Maintainability Comparison Summary (Clock Configuration App) .	94
6.6	Interaction Latency Comparison (Clock Configuration App, 50ms network RTT)	95
6.7	Development Workflow Comparison	96
6.8	RAP Exclusion Decision Matrix	97
6.9	Performance Comparison: Clock Configuration Tool (Standard Business Application)	99
6.10	Rendering Performance Comparison: Automotive Bus Editor (High-Frequency Updates)	99
6.11	Strategic Pathway Selection Criteria: React vs. Slint/Wasm	100
6.12	Strategic Pathway Comparison: Empirical Findings Summary	101

7.1	Quantified Cost-Benefit Summary	111
-----	---	-----

Acronyms

ACS	Aurix Configuration Studio.
AJAX	Asynchronous JavaScript and XML.
AOT	Ahead-of-Time.
API	Application Programming Interface.
CD	Continuous Delivery.
CI	Continuous Integration.
CLI	Command-Line Interface.
CAD	Computer-Aided Design.
CAN	Controller Area Network.
CNCF	Cloud Native Computing Foundation.
CPU	Central Processing Unit.
CSP	Content Security Policy.
CSS	Cascading Style Sheets.
DAP	Debug Adapter Protocol.
DI	Dependency Injection.
DOM	Document Object Model.
DSR	Design Science Research.
EA	Enterprise Architecture.
EMF	Eclipse Modelling Framework.
FFI	Foreign Function Interface.
FPS	Frames Per Second.
GC	Garbage Collection.
GEF	Graphical Editing Framework.
GLSP	Graphical Language Server Platform.
GMF	Graphical Modelling Framework.
GPU	Graphical Processing Unit.
GUI	Graphical User Interface.

HMI	Human-Machine Interface.
HTML	HyperText Markup Language.
HTTP	HyperText Transfer Protocol.
IDE	Integrated Development Environment.
INP	Interaction to Next Paint.
IPC	Inter-Process Communication.
ISO	International Organization for Standardization.
JEE	Java Enterprise Edition.
JIT	Just-In-Time.
JVM	Java Virtual Machine.
JSON	JavaScript Object Notation.
LIN	Local Interconnect Network.
LOC	Lines of Code.
LSP	Language Server Protocol.
MI	Maintanability Index.
NPM	Node Package Manager.
OS	Operating System.
OSGi	Open Service Gateway Initiative.
RAP	Remote Application Platform.
RCP	Rich Client Platform.
REST	REpresentational State Transfer.
RPC	Remote Procedure Call.
RWT	Remote Widget Toolkit.
SCM	Source Control Management.
SQALE	Software Quality Assessment based on Lifecycle Expectations.
SRAPA	Spring RAP Automation.
SWT	Standard Widget Toolkit.
TCO	Total Cost of Ownership.
TDR	Technical Debt Ratio.
TTI	Time to Interactive.
UI	User Interface.
URI	Uniform Resource Identifier.

URL	Uniform Resource Locator.
UX	User Experience.
VFS	Virtual Filesystem.
VS Code	Visual Studio Code.
WAI	Web Accessibility Initiative.
WAR	Web Application Archive.
Wasm	WebAssembly.
WCAG	Web Content Accessibility Guidelines.
WebGL	Web Graphics Library.
WebGPU	Web Graphics Processing Unit.
XML	Extensible Markup Language.
XSS	Cross-Site Scripting.

Chapter 1

Introduction

The development of software architecture in the enterprise sector is distinguished by an unstoppable transition from conventional thick client apps to distributed, cloud-based environments. For organisations, this shift poses serious difficulties that depend on complex desktop platforms, such as those based on the Eclipse Rich Client Platform (RCP). This thesis examines the technical, architectural, and economic effects of updating outdated Eclipse RCP applications, evaluating critical tactics and newly developed replacement platforms that enable a smooth transition to the web with a prospective orientated to the cloud. This first chapter provides background information, describes the architectural challenges intrinsic to the migration process, explains the strong reasons for change, and describes the methodology used in this study.

1.1 Background and Contextual Foundation

1.1.1 Eclipse Rich Client Platform (RCP) and Domain-Specific Tooling

The Eclipse Rich Client Platform stood out nearly two decades ago as one of the main go-to technologies for the development of extensible desktop applications, tools, and custom Integrated Development Environments (IDEs). Eclipse RCP has achieved great success and has been widely adopted in different contexts and environments over the years, ranging from engineering environments and modelling tools to mission-critical control centres [1]. Its success within specialised domain tools is mainly due to the stability, open specifications, and vendor-neutral governance offered under the Eclipse Foundation. The platform architecture is built on the **OSGi** framework, which, thanks to its high modularity and extensible design, provides a proven foundation for large and complex applications, ensuring a clear separation of components, robust version control, and secure extensibility [2]. The application lifecycle within Eclipse RCP is defined by the extension point

"org.eclipse.core.runtime.applications", which requires the implementation of the "IApplication" interface, which acts as the entry point equivalent to a traditional

Java `main()` method [3]. Additionally, RCP uses the **Standard Widget Toolkit (SWT)**, which provides a desktop user experience by allowing native rendering on platforms [4]. However, this intrinsic connexion to native desktop libraries and local resources also defines the structural limitations now facing modernisation efforts.

1.1.2 The Foundational Limits of the Thick Client Model

The architecture of Eclipse RCP applications is inherently desktop-centric, imposing limits on agility and deployment. Specifically, its orientation towards single-machine execution and non-distributed processing is increasingly difficult to align with modern development needs [1]. Applications based on thick clients were historically justified when computation was intensive per transaction or involved significant local data transfer [5]. However, as enterprise needs shift towards highly reliable, horizontally scalable, and distributed systems, the local execution model of Eclipse RCP applications has become a primary bottleneck to scalability and resilience.

1.2 Motivation for Modernisation: Economic and Strategic Drivers

The primary reason for migrating away from legacy rich client platforms is no longer rooted solely in feature disparity, but is driven by compelling shifts in economic realities, user expectations, and the competitive landscape of the software industry.

1.2.1 The Critical Shift in User and Business Expectations

Modern user experience (UX) has fundamentally evolved from the past, moving away from desktop installation towards friction-free, standardised access. The browser has become the universal interface [5]. Users now expect clean and responsive user interfaces, smooth animations, and cross-device support [1]. This contrasts sharply with the often rigid and specialised nature of legacy desktop tooling.

For businesses, the appeal of modernisation is rooted in delivery models. Modern enterprises prioritise **friction-free onboarding**, where a simple URL eliminates the need for complex local toolchains, dependencies, and IT administration typically associated with thick clients. Furthermore, business models have embraced Software-as-a-Service (SaaS) subscriptions and cloud workspaces, which natively support centralised, easy updates and are essential for today's distributed, remote-first work environments [1]. Maintenance of thousands of different desktop environments, each with its own operational quirks, interference issues, and upgrade requirements, creates a significant support burden that web-based deployment eliminates [5].

1.2.2 Total Cost of Ownership (TCO) Escalation and Talent Scarcity

The long-term viability of legacy architectures is severely hampered by escalating Total Cost of Ownership (TCO), driven primarily by accumulated technical debt and a contracting talent pool. Technical debt represents future costs resulting from suboptimal decisions or shortcuts made during development, which accumulates "interest" over time [6]. This debt manifests itself as an increase in complexity, making maintenance progressively more time-consuming and expensive [7].

The issue is compounded by a **talent crisis**. The ecosystem of developers experienced and willing to work with the legacy Eclipse stack, particularly SWT, OSGi, and Eclipse RCP, is declining [1]. This scarcity drives up the cost of specialised skills, contributing to the increasing maintenance burden and making the technology platform unattractive to the top new talent [7]. Consequently, specialised technical debt converts directly into financial and organisational liability.

Furthermore, the continuous migration of other stakeholders away from the Eclipse Platform initiates a negative feedback loop. As the pool of shared investment, maintenance, expertise and innovation shrinks, the cost and risk profile of the remaining adopters increases [1]. This erosion of the ecosystem accelerates the obsolescence curve and makes strategic investment in the legacy platform increasingly untenable.

1.2.3 The Architectural Imperative of Cloud-Native Adaptation

The move to cloud-native computing is an architectural necessity for organisations seeking exceptional agility and scalability. Legacy applications struggle to provide the resilience and resource management required to handle fluctuating enterprise workloads. Cloud environments, on the contrary, provide dynamic resource management and the ability to rapidly adjust the capacity of the IT system based on current operational conditions [8].

Enterprise architecture (EA) must shift its focus from static system alignment to enabling speed, scalability, and resilience [9]. This requires migrating from tightly coupled monolithic applications to modular and composable architectures that can be adopted and governed autonomously by individual teams. Legacy Eclipse RCP structures, often characterised by intricate dependencies of plugins and complex configuration, actively hinder the adoption of modern DevOps methodologies, automated CI/CD pipelines, and native cloud security measures [10].

The comparative drivers for architectural choice are summarised below:

Cost Factor	Legacy Eclipse RCP	Modern Web/Cloud-Native	Academic Rationale
Deployment and Updates	High (Local installs, OS-specific quirks) [5]	Low (Centralised, URL-based access) [1]	Reduces Process Debt and organisational friction.
Maintenance Burden	High (Accumulation of technical debt, difficult architecture)	Low (Modularity allowed, standardised frameworks)	Technical debt translates into financial liability (TCO).
Talent Pool	Specialised and Declining [1]	Large and Mainstream [1]	The largest long-term economic risk to legacy systems is talent scarcity [7].
Scalability Model	Vertical/Limited (Single-machine execution) [1]	Horizontal/Elastic (Cloud-Native design) [11]	Essential for responding to modern workload variations and global data distribution [8].

Table 1.1: Comparative Analysis of TCO Drivers in Application Architectures

1.3 Issues and Architectural Challenges of Eclipse RCP Migration

Despite the clear motivation, migrating an established Eclipse RCP application is fraught with specific technical impediments rooted in architectural and process debt.

1.3.1 Technical Debt Manifestations Specific to RCP

Eclipse RCP applications often harbour specific technical debt that complicates modernisation efforts. A notable source of process debt is the reliance on extensive, manually maintained XML configuration files, such as manifest dependencies and target definition files. The necessity of manually updating sequence number elements in XML configuration files upon adding a dependency can lead to confusing and random compile errors if forgotten. These archaic configuration practices contrast sharply with the streamlined dependency management of modern toolchains.

Furthermore, the architectural design of legacy core OSGi, bundles, such as "org.eclipse.core.resources", exhibits poor separation between API and implementation. This tight coupling makes surgical refactoring exceptionally difficult. Performance optimisations are another prevalent issue. Migrations to newer Eclipse versions have resulted in severe degradation in start time, increasing launch times from three seconds to 17 seconds in reported cases. Mitigating these issues often requires complex programmatic workarounds, such as explicitly disabling them through runtime options such as "cssTheme none", consuming valuable development resources merely to restore acceptable performance [12].

As the pool of developers willing to work with legacy technologies such as SWT and OSGi shrinks, organisations face a strategic imperative to define a modernisation path that varies web technologies [1].

Legacy System Characteristics	Observed Values
Typical age of legacy applications	15-20 years
Legacy systems lacking proper documentation	67%
Applications using unsupported languages/frameworks	91%
Legacy apps with monolithic databases	72%
Automated test coverage in legacy apps	23%

Table 1.2: Legacy Application Age and Modernisation Challenges [13]

1.3.2 The Migration Paradox: Rewrite vs Reuse

A critical realisation for enterprise stakeholders is that moving to a web-based platform is not analogous to a typical upgrade of the Eclipse version; it requires a fundamental architectural pivot. Crucially, all application user interfaces built using platform-specific components such as SWT and JFace must be completely re-implemented using modern web frameworks such as React, Angular, or Vue [14]. This UI rewriting mandate introduces significant risks.

A full rewrite, while offering the benefit of a clean, modern foundation, can take years to achieve feature parity with a mature legacy product. This process is expensive, risks introducing regressions, and often leads to developer fatigue and churn, especially when re-building functionality that already works [4]. The central architectural problem for modernisation strategies is therefore defined by the decoupling challenge: how to minimise the high-risk, high-effort UI rewrite component while maximising the reuse of the stable, core Java business logic.

Failing to evaluate all the trade-offs associated with reuse can lead to serious problems within architecture. The way architects and developers discover candidates for reuse is through **abstraction**. Observing that some reuse cause brittleness begs the question of how that kind of reuse differs from the kinds we clearly benefit from. Operating systems, open source frameworks, and libraries are all things that everyone successfully reuses. The **rate of change** is what really distinguishes these from the assets that projects teams build. Technical coupling brings about a slow rate of change or a well-understood rate of change and update cadence, which benefits developers and architects. Internal domain capabilities or rapidly changing technical frameworks make terrible coupling targets [15].

1.4 Emerging Eclipse Migration Targets for Domain-Specific Tooling

The Eclipse ecosystem has produced several platforms designed to serve as successors to the traditional Eclipse RCP for complex tooling, each of which presents a distinct architectural profile for the migration effort.

The choice of a target platform involves a fundamental trade-off between maximising Java code reuse and achieving genuine cloud-native architectural efficiency. This trade-off is often defined by the User Interface (UI) rendering model.

1.4.1 Visual Studio Code: The Dominant Extension Ecosystem

Visual Studio Code (VS Code) has emerged as the dominant code editor platform, with its extension API becoming a de facto standard in developer tooling [16]. Built on Electron, Chromium and TypeScript, VS Code provides a rich Extension API that allows developers to add language support, debuggers, and custom UI components [17]. Its massive extension ecosystem (more than 40,000 extensions) and native support for LSP and DAP make it an attractive migration target for Eclipse-based tools focused on code editing and language tooling [18].

However, VS Code's architecture imposes significant constraints for complex Eclipse RCP migrations. The extension API is deliberately restricted—extensions cannot fundamentally alter the workbench layout, create custom perspectives, or deeply integrate with the behaviour of the core editor in ways that Eclipse RCP plugins routinely do [19]. Custom graphical editors, model-driven development tools, or applications requiring extensive workbench customisation find VS Code's extension model too restrictive.

VS Code is therefore best suited for migrating Eclipse-based tools that align with its "editor-centric" paradigm—language tooling, debuggers, and code-focused workflows—rather than comprehensive workbench applications that require deep customisation and complex user interface layouts [20].

1.4.2 Eclipse Theia: The Cloud-Native Workbench Successor

Eclipse Theia is positioned as the natural web and cloud migration path for complex, legacy Eclipse Platform adopters, particularly those running "Workbench applications" that require features like a stacked view layout, a plugin/extension system, and access to an underlying file system [21]. Theia is not a direct upgrade but a completely new technical foundation built on modern web technologies (TypeScript and Node.js) [22]. Theia's architecture is inherently distributed, running in two separate processes: a frontend (client) process rendering the UI (via browser or Electron) and a backend (server) process running on Node.js [23]. These processes communicate via JSON-RPC messages over WebSockets or REST APIs, making it highly suitable for cloud deployment [23]. This architectural choice enables truly distributed processing and superior scalability compared to desktop-first models. Theia's support for the VS

Code extension ecosystem, via Open VSX, provides a significant strategic benefit, moving adopters from the shrinking Eclipse RCP plugin base to the dominant, growing ecosystem in developer tooling [22]. Case studies, such as the re-platforming of Martini Designer from Eclipse RCP to Theia, show its ability to unify visual editors and coding support in a modern and versatile environment [24].

1.5 Modernisation Methodologies

Successful migration requires a strategic approach that acknowledges the risk of a complete rewrite and the technical debt of a more conservative approach, by focussing on disciplined architectural decoupling and incremental modernisation.

1.5.1 Strategic Pathways and Application Modernisation Models

Modernisation strategies are often categorised using the "6 R's" framework. For complex domain-specific desktop applications such as RCP tools, simple **Re-hosting** (lift and shift) is generally ineffective, as it fails to take advantage of cloud benefits [25]. Instead, viable pathways involve deeper modification: **Re-platforming** (re-tooling the application to optimise for web and cloud features without changing core architecture) or **Re-architecting** (a comprehensive redesign, often breaking down monolithic structures into microservices) [10].

The most critical initial step, regardless of the ultimate target platform, is **Refactoring** the existing system to achieve architectural decoupling. This process extracts the domain logic into headless and platform-agnostic components, accessible through standard interoperable interfaces such as CLI, REST services, or the Language Server Protocol (LSP). Separating business logic from UI liability maximises component reuse, allowing a "single source" where the same core component can serve both the existing Eclipse RCP application and the new web solution simultaneously, driving cost efficiency [14].

1.5.2 Incremental and Transitional Strategies

The transition phase can be systematically controlled by employing strategies that reduce risk and preserve operational continuity. The **Strangler Fig Pattern** is particularly advantageous in this context, as it enables newly modernised, web-based functionalities to be integrated into the existing Eclipse RCP application via the SWT embedded browser widget. In this way, updated components can be continuously integrated and validated before a full cutover is performed.

In addition, introducing standardised protocols establishes an essential abstraction layer. The **Language Server Protocol (LSP)** has facilitated uniform language tooling across different IDEs [14]. Building on this idea, the **Graphical Language Server Platform (GLSP)** defines a protocol specifically for diagram editors [26]. By relocating domain-specific language (DSL) or modelling logic to an LSP/GLSP-based server, the same core capabilities can be provided in parallel within both the legacy Eclipse desktop environment and the new web-based application, thereby preserving behavioural consistency while lowering migration risk [14].

1.5.3 Eclipse RAP

Eclipse RAP (Remote Application Platform) aimed to bridge this gap by allowing developers to build web applications using the familiar Java/SWT API.

Eclipse RAP operates on a server-side UI model: SWT components are rendered on the server, and the UI interaction is relayed to the client via an AJAX/JavaScript layer. While RAP achieves very high reuse of Java code, minimising initial rewrite effort, its architecture suffers from limitations in true cloud-native contexts. In contrast, platforms such as Theia require a UI rewrite but offer a client-side UI model. This design shifts rendering and much of the responsiveness burden to the user's browser, drastically reducing server load, improving network latency, and optimising performance for cloud-based multi-user concurrency [27].

1.5.4 Eclipse GLSP: The Future of Diagram Editors

Graphical editors are a specialised component of many Eclipse RCP tools (often built using GEF or GMF). The Eclipse Graphical Language Server Platform (GLSP) is the intended replacement, providing an open-source framework for building modern, web-based diagram editors [28]. GLSP separates the client (rendering using HTML5, TypeScript, and SVG) from the graphical language server (which handles loading, interpreting, and editing rules) via a defined protocol. This protocol-based architecture allows the server-side business logic (often using models such as EMF) to remain backend-agnostic and enables the diagram client to be directly embedded in various frontends, including Eclipse Theia, VS Code, or stand-alone web applications [26]. Although GLSP effectively facilitates the migration and modernisation of highly complex, domain-specific graphical editors, it offers no architectural pathway for migrating the vast majority of a standard Eclipse RCP application. A typical Eclipse RCP application consists of a comprehensive workbench structure, including perspectives, views, menu structures, property sheets, and standard text or table editors, all built upon the SWT/JFace API [29].

1.5.5 Equo SWT Evolve: The "Lift-and-Shift" Bridge

Equo SWT Evolve focusses on modernisation in place. It serves as a drop-in replacement for the SWT library, offering a modern desktop UI with GPU acceleration without requiring code changes to the use of the SWT API. Although this approach offers rapid desktop modernisation and even single-user web deployment, it retains the core Java/SWT stack, meaning that the fundamental architectural limitations related to distributed multi-user cloud scalability are not addressed, making it primary a short-to medium-term solution for UI/UX improvement rather than a full cloud migration path [4].

1.5.6 Web Technologies

Although Eclipse RAP, GLSP, and Equo each address specific aspects of modernisation, migrating to a truly web-based architecture requires adopting a fundamentally different technology stack. Modern web platforms such as Eclipse Theia, VS Code for the Web, or custom web applications are built upon a set of standardised web technologies that replace the Java/SWT/OSGi foundation of Eclipse RCP.

TypeScript and JavaScript have become the lingua franca of web-based tool development. TypeScript, a statically-typed superset of JavaScript, provides type safety and object-oriented patterns familiar to Java developers while embracing the asynchronous, event-driven nature of web development [30]. The major IDE platforms including VS Code, Theia, and the web-based components of JetBrains Fleet are all built primarily in TypeScript, establishing it as the de facto standard for this domain.

Frontend frameworks, for example, React, Angular, or Vue.js, substitute SWT/JFace’s widget toolkit with component-driven architectures. In contrast to SWT’s imperative approach to widget instantiation, these libraries use declarative UI definitions, in which components specify what should be rendered based on the current application state [31]. Dedicated libraries extend these frameworks themselves: Monaco Editor delivers sophisticated text-editing functionality comparable to Eclipse’s text editors, while Phosphor and similar widget toolkits enable dockable panels, command palettes, and other UI patterns typical of IDEs [32].

However, this migration path needs substantial funding. The entire UI layer must be rebuilt using web technologies, requiring teams to master HTML5, CSS, modern JavaScript frameworks, and web-specific build tools (npm, webpack, ESLint) that differ essentially from Eclipse’s PDE and OSGi-based development environment [33]. Unlike RAP’s code reuse or Equo’s drop-in replacement, web-based migration offers no shortcut—it requires a complete architectural transformation. The benefit is a truly modern browser-based application with outstanding performance, broad platform compatibility, and conformity with contemporary development practices, but the cost is substantial development effort and organisational change [34].

Tool	Primary Strategy	UI Rendering Model	Java Core Logic Reuse	Cloud-Native Efficiency
Web Technologies	Re-architecting UI	Client-Side (TypeScript/React/WebAssembly) [22]	High (Decoupling via REST/LSP) [14]	Very High (Supports distributed processing/scalability) [23]
Eclipse RAP	Re-platforming UI	Server-Side (SWT emulation) [27]	Very High (Retains SWT API calls) [27]	Moderate (Limited concurrency due to server load)
Equo SWT Evolve	In-place Modernisation	Client-Side (GPU-Accelerated) [4]	Near 100% (Drop-in replacement) [4]	Low (Generally limited to single-user deployment model) [4]

GLSP (Diagram- ming)	Component Refactor- ing	Client-Side (HTML5/SVG) [26]	High (via Proto- col Server) [35]	High (Protocol- based decou- pling maximises reuse)
-------------------------------------	-------------------------------	------------------------------------	--------------------------------------	--

Table 1.3: Architectural Evaluation of Eclipse Migration Target Platforms

1.6 Case study: Microcontroller Tooling in the Automotive industry

Software development tools are rapidly moving to the cloud and are often required on browser-based platforms. Integrated development environments (IDEs) were once mainly native desktop applications, with a deep connexion to the native system and OS. Today, they are often built using web technologies such as HTML, CSS, JavaScript, or TypeScript. This shift is happening because organisations are increasingly seeking more reproducible environments, enhanced security, and easier collaboration tools. Progress made in the research of containerisation, remote development protocols, and language server standards is also making this change possible. As teams start using web-based IDEs like Visual Studio Code, Eclipse Theia, and hosted options such as GitHub Codespaces, they face a significant challenge: migrating existing extensions and plugins, which were designed for native environments, to web-first platforms without compromising functionality, performance, or developer experience. Extensions are at the core of modern IDEs. They provide and facilitate language support through the Language Server Protocol (LSP), allow debugging with the Debug Adapter Protocol (DAP), enable the creation of custom views and interaction points with the end-user, integrate with external services, and automate tasks. Over the last decade, extension ecosystems have evolved around native features, including complete file system access, running local processes, utilising local toolchains, deep operating system integration, and desktop UI frameworks. However, Web-based IDEs work in more limited environments. The extension host might be a Web Worker; file system access is often virtual or managed by proxies; creating new processes is limited; and user interfaces must use web or declarative APIs. Electron-based IDEs use web technologies but typically provide more system access than browser-only IDEs. Embedded systems development has been stuck on native platforms for many years, and is probably one of the most difficult fields to migrate completely, due to the challenge of moving to a web-based environment without renouncing native-speed.

1.6.1 Infineon Technologies and Aurix Configuration Studio (ACS)

The idea behind this thesis stems from a collaboration with Infineon Technologies, aimed at examining the transition from native IDEs to web-based platforms, with a focus on how extensions are migrated between them. Infineon Technologies is a semiconductor manufacturing and design company. In addition, they develop internal software tools and IDEs to facilitate the development and debugging of embedded systems, mainly in the automotive industry.

ACS is the customised IDE developed by Infineon Technologies. The primary objective is to simplify the configuration of modular software components, enabling developers to initiate AURIX application development more efficiently by providing an intuitive and automated solution to configure both software components and AURIX devices. The ACS' graphical user interface enables users to configure and customise projects

intuitively, reducing the complexity compared to traditional development methods. The automated resource management framework, driven by an AI-based solver, automatically assigns and maintains hardware resources, freeing developers from repetitive tasks that take up a lot of time. Based on the interface settings, the ACS' tools automatically generate high-quality production-ready code, minimising manual coding errors.

The AURIX Configuration Studio is based on Infineon's DAVE technology. ACS's IDE integrates code generation and debugging tools, as well as a wide range of software components such as drivers, middleware, and libraries optimised for Infineon's hardware. Building on established technology, Infineon can offer AURIX developers a robust and reliable software development solution with ACS.

ACS offers many applications to the end-user that assist in the development of the AURIX components. These are Eclipse RCP applications that are developed and then implemented inside ACS, without the need for an external installation. They strongly rely on the Eclipse Equinox runtime and cannot be used in a stand-alone version.

1.7 Structure of the Thesis and Original Contribution

The preceding sections establish that the modernisation of Eclipse RCP applications is an economic and strategic need, driven by rising TCO, talent scarcity, and the need for cloud-native scalability. However, this necessity is constrained by the architectural friction inherited from the legacy thick client model, particularly the tight coupling between the user interface and the core business logic.

1.7.1 Research Questions and Hypotheses

The overarching goal of this thesis is to investigate the strategic relationship between architectural decoupling efforts and the resultant migration efficiency to modern, web-based platforms. Key research questions include:

1. How can legacy Eclipse RCP applications be effectively refactored to minimise technical debt while maximising business logic reuse?
2. What are the quantifiable compromises between migration pathways prioritising Java code reuse versus those requiring UI rewrite for cloud-native optimisation?
3. Which modern web technology stack is best suited for a full UI overhaul in embedded systems tooling?
4. Can a strategic modernisation framework help enterprise architects determine the most suitable migration strategies for domain-specific tool ecosystems?

1.7.2 Overview of the Research Methodology

This research employs a mixed-method approach. The initial phase involves an extensive review of the literature and a comparative architectural analysis, which lay the theoretical foundation and quantifies the drivers of technical debt. The analysis will compare the results of applying different modernisation strategies (e.g., Eclipse RAP) against projected full-rewrite scenarios, providing evidence-based conclusions on effort and risk. This is followed by an empirical validation phase using two technical case studies: the first involves the modernisation of a representative Eclipse RCP application; the second one highlights the limitation of this platform and the advanced possibilities of stepping out. Established software metrics will be used to quantify migration effort, including rewritten code lines, reduced complexity, and post-migration performance improvements related to startup time and server load.

1.7.3 Original Contribution and Thesis Structure

The original contribution of this thesis lies in the development and validation of a nuanced strategic framework tailored specifically for enterprise decision-makers and architects managing proprietary Eclipse RCP applications. This framework will offer clear, data-driven guidance on balancing the conflicting objectives of maximum Java code reuse and achieving the necessary efficiencies of true web-based and cloud-ready

architecture.

The subsequent chapters of this thesis are structured as follows:

- **Chapter 2: Literature Review and Architectural Analysis** provides a detailed examination of different platforms, modern web technologies, and quantified technical debt models.
- **Chapter 3: Methodology and Design** details the research design, architectural metrics, and the implementation process for the modernisation experiments.
- **Chapter 4: Implementation of Modernisation Pathways and Integration Frameworks** details the technical implementation of three migration pathways: Eclipse RAP, React, and WebAssembly with Slint.
- **Chapter 5: Integration with IDE Ecosystem** presents a dedicated section on Extension Orchestration, describing the architectural design and asynchronous message-passing protocols used to embed these pathways into VS Code via Webview containers and Eclipse Theia via Theia extensions.
- **Chapter 6: Empirical Analysis and Comparative Evaluation** applies the previously developed implementation pathways to two different empirical case studies: a conventional form-based tool and a complex, resource-intensive graphical application.
- **Chapter 7: Conclusion and Prospective Work** recaps the principal contributions and outlines prospective avenues for future investigation in the domain of legacy tool modernisation.

Chapter 2

Literature Review and Architectural Analysis

The software engineering landscape is currently defined by a decisive shift from a localised desktop-centric environment to ubiquitous web-based delivery. For more than two decades, the Eclipse Rich Client Platform (RCP) has served as the standard for mission-critical engineering and industrial tools, leveraging the robustness of the Standard Widget Toolkit (SWT) and the modularity of OSGi (Open Service Gateway Initiative). However, the rise in expectations for modern user experience (UX) has placed traditional Eclipse RCP architectures in a state of increasing obsolescence [1]. This chapter examines the platform architectures, technologies, and web principles required to transition legacy Java logic into modern browser-based environments.

2.1 Comparative Analysis of Platform Architectures

Transitioning from desktop to web requires a deep understanding of the structural differences between classic Java frameworks and modern IDE platforms.

2.1.1 Classic Eclipse RCP: The OSGi Monolith

The longevity of Eclipse RCP is rooted in its innovative plugin-based architecture, which used the Equinox implementation of OSGi to manage the life cycle and dependencies of modular software components. This model allowed teams to build highly extensible, domain-specific applications such as space mission control centres and complex engineering simulators. However, the core architectural assumptions of Eclipse RCP (desktop-first, non-distributed, and orientated towards local file system access) are increasingly incompatible with modern web paradigms. The Standard Widget Toolkit (SWT) and JFace provide high performance by mapping Java calls directly to the host operating system controls, but this tight coupling prevents native browser delivery without significant translation layers [1].

Architecturally, Eclipse RCP is a stateful, single-process application running on a single Java Virtual Machine (JVM). Although highly modular, its user interface

layer is designed for a single-user desktop windowing system. This model creates "architectural friction" with cloud paradigms because the user interface (SWT) is tightly coupled to the local operating system's windowing system, making native browser delivery impossible without translation layers [36].

Eclipse RCP is a modular platform for building native desktop applications in Java. It provides a proven runtime (Equinox/OSGi), a rich UI stack (SWT/JFace/e4 workbench), declarative extensibility, and a complete provisioning/update system. Unlike "the Eclipse IDE," which is a specific application built on the platform, Eclipse RCP is the underlying technology that allows the development of branded stand-alone applications with the same foundations: modularity, cross-platform user interface, and industrial-strength lifecycle and update capabilities. Organisations adopt Eclipse RCP when they need:

- Deeply integrated desktop tools or engineering workbench with long life cycles.
- Strong modularity and plugin ecosystems with strict versioning and API discipline.
- A native cross-platform user interface without the need to rewrite for each operating system.

2.1.2 Eclipse RAP: The Headless Workbench

"The Remote Application Platform (RAP) project aims to enable developers to build rich Ajax-enabled Web applications using the Eclipse development model, plug-ins and a Java-only API." [37].

The Remote Application Platform enabled single-sourcing by allowing Java/SWT code to run in a web browser. It uses a client-server protocol, where the UI logic stays on the server and is rendered via a thin Ajax-based client. To create UIs, RAP uses the Java API of SWT, which is provided by the RAP Widget Toolkit (RWT). RAP applications are automatically rendered in the browser as standard web applications. However, there is no guarantee of a one-to-one mapping between SWT widgets and HTML tags. The mapping is internal and depends on various details: RAP version, theme, client implementation. There is no precision in foreseeing the generated HTML output and tags, and thus handling the widgets' properties via DOM manipulation.

The flexibility of RAP is shown by the various configuration possibilities, described in Tab.2.1. Eclipse RCP applications are tightly integrated with the Eclipse Workbench and rely on it. One of the goals of this thesis is to try to abstract the application from the constraints of the OSGi environment.

	RAP with OSGi	RAP with Workbench	RWT Standalone
Summary	Lightweight and flexible, recommended for new applications that do not use Eclipse extensions	For applications using the Eclipse Workbench (RCP applications)	No OSGi, RWT used as a library in a traditional web app [38].
OSGi	Compatible with any OSGi implementation including Equinox	Eclipse Equinox required	No OSGi
Deployment	As WAR file in servlet container or directly in OSGi container with embedded server. Direct access to JEE contexts possible with JEE compatibility mode.	As WAR file in servlet container or directly in OSGi container with embedded server.	As WAR file in servlet container. Direct access to JEE contexts possible with JEE compatibility mode. Clustering supported [38].
Configuration	Using the Application Configuration API	Using extension points Branding	Using the Application Configuration API RWT servlet and application configured in web.xml [38].
Entry-point	EntryPoint	EntryPoint or IApplication	EntryPoint
Operation Modes	SWT or JEE compatibility mode	SWT compatibility mode enforced	SWT or JEE compatibility mode [38]
RCP Compatibility	SWT, JFace	SWT, JFace, Workbench	SWT, JFace
Launch from IDE	RAP Launcher	RAP Launcher	RWT Launcher

Table 2.1: RAP setups

In the context of Eclipse RAP, we define an application configuration as the set of settings that create and start an application instance. It contains various components, including entry points, URL mappings, themes, and service handlers. An **Entry Point** is a Java class that extends the abstract class "AbstractEntryPoint". The

only abstract method that needs to be implemented must contain all user-interface content written using the Java API of SWT and be rendered by RAP as a web application. An application can have one or many entry points, each mapped to a separate servlet path.

The **RAP client** serves as a JavaScript-based RWT runtime and toolkit that operates within a web browser. Delivers the UI to the user and captures the input; then transmits the events (such as typing and clicking) to the RAP server using the RAP Protocol [39]. The **RAP Protocol** is used to describe the JSON-based message format used in HTTP communication between an RAP client and an RAP server. With RAP 2.0, all communication between the client and the server uses this protocol [40]. The **RAP server** lives in a Java web container as a servlet. It runs the UI content written in Java and sends back the UI updates to the RAP client.

2.1.2.1 RAP Threads

Eclipse RAP keeps the SWT single-threaded UI programming model, but implements it over HTTP requests. All user events are sent from the browser to the server, processed in one “UI thread” per user session, and the resulting UI changes are sent back to the browser. Background work runs on regular threads/jobs and must request back to the UI thread to touch widgets. For each user session, there is a single UI context (Display). Only one thread can access it at a time. Eclipse RAP does not maintain a long-lived dedicated user interface thread. Each HTTP request is processed by a worker thread that “becomes” the UI thread for the duration of that request. Requests for a session are serialised, so there is no concurrent access to widgets. To update the UI from the background work, the methods "Display.asyncExec(Runnable)" or "Display.syncExec(Runnable)" are used. If there is no client request in progress, the queued UI work will only run when the client follows up and talks to the server. The "syncExec" method from a background thread blocks until the runnable has executed on the UI thread, which requires a request. Without a server push (or a user request), it can be blocked indefinitely.

2.1.3 Visual Studio Code: Multi-Process Sandbox

Visual Studio Code (VS Code) is a free, open-source, and cross-platform code editor. It is fundamentally a modular text editor that offers powerful development features through extensions and standardised protocols.[20]

VS Code employs a distinct, multi-process architecture based on Electron, in which the application is divided into three main categories of processes:

- **Main Process:** Coordinates the overall application lifecycle and oversees integration with native operating system features.
- **Renderer Process:** Manages the Chromium-driven user interface (Workbench), rendering it through HTML, CSS, and TypeScript.

- **Extension Host:** An isolated Node.js process in which extensions run asynchronously [41].

This design ensures that a crash in an extension does not compromise the entire editor, maintaining UI responsiveness. However, the VS Code extension API is deliberately constrained to a simple tree-like dependency graph, which limits the ability of developers to customise the core workbench layout or remove default features.

VS Code is primarily built using web technologies and works on various platforms:

- **Desktop:** Everything runs on the user machine. This setup gives the fastest feedback and is the easiest to set up.
- **Desktop with remote:** The user interface stays on the machine, but the main work—such as extensions, language servers, and tools runs where the code and runtime are located. This setup enables faster file operations and language features while also maintaining consistency across tools.
- **Browser:** The UI runs within the browser; compute runs on a remote server (Codespaces) or is limited to browser capabilities (vscode.dev). Web-ready extensions and cloud file providers fill in the gaps.

The desktop version of VS Code is built with Electron. Electron is an open-source framework that enables developers to build desktop applications for various platforms using web technologies such as HTML, CSS, and JavaScript [42]. Provides the following:

- A main process (Node.js) that manages windows, menus, updates, and OS integration.
- One or more renderer processes (based on Chromium) are responsible for displaying the user interface. The VS Code “workbench” itself is implemented as a web application that is rendered within these processes.
- A shared or background process handles services like telemetry and extension management. There are also extra helper processes, such as one for the terminal.

The desktop IDE utilises Electron IPC for inter-process communication (IPC) between the main process and the renderer, as well as message ports or sockets for communication between the renderer and the extension host. The remote version uses SSH or web sockets to multiplex requests.

By offering separated components, VS Code provides process isolation: UI, extension host(s), and sometimes language servers each get their own process, allowing for failures while keeping the UI simple.

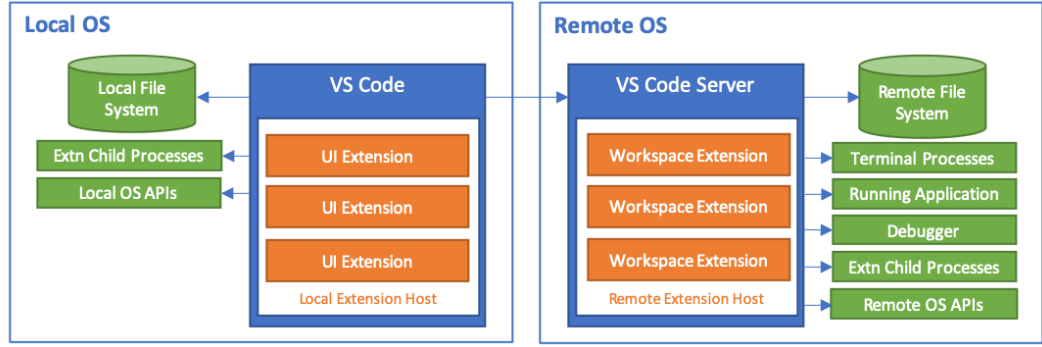


Figure 2.1: VS Code extension host architecture [43]

2.1.3.1 Extension manifest

Each VS Code extension needs a manifest file called "package.json". It is located at the root of the extension directory structure. It contains various fields used to manage the extension and its properties. Some of them are:

- **Contribution Points:** JSON declarations in the contributes field of the package.json extension manifest. Extensions register Contribution Points. This enables them to extend various features in the VS Code.
- **Activation Events:** JSON declarations in the "activationEvents" field of the package.json extension manifest. These allow lazy activation of extensions, which can be started only when specified to reduce consumed resources.

2.1.3.2 Views

Views contain content and can appear in the Sidebar or Panel. There are three primary types: Tree Views (which display hierarchical data), Welcome Views (which offer introductory or instructional content), and Webview Views (which present custom web-based interfaces). Users can rearrange Views or move them between Containers, such as from the Primary Sidebar to the Secondary Sidebar. View Containers serve as the primary locations for displaying Views. Extensions can add custom View Containers to the Activity Bar, Primary Sidebar, or Panel. Users may move an entire View Container from the Activity Bar to the Panel or vice versa, and can also relocate individual Views. An example of a View Container is the File Explorer view, which enables users to browse, open, and manage files and folders within a project.

Views can be added to existing View Containers, such as File Explorer, Source Control (SCM), or Debug, or to a custom View Container via the Activity Bar or any Panel View Container.

2.1.3.3 Webview API

The Webview API's flexibility allows for great freedom in the realisation of extensions, opening a new and powerful way of customisation. A webview functions similarly to

an iframe within VS Code, allowing extensions to control its content. It can render nearly any HTML content and communicates with extensions through message passing [44].

Although they offer great personalisation, webviews should be used carefully because they require a substantial amount of resources. Additionally, webviews operate in an isolated browser environment, which can affect how they interact with other extension components that run on the Extension Host.

2.1.3.4 Message Passing with the JSON-RPC protocol

JSON-RPC is a lightweight remote procedure call (RPC) protocol encoded in JSON that eases the communication between VS Code extensions and webviews. VS Code often uses JSON-RPC to enable interactions between language servers and language clients, as well as between extensions and external processes [45].

- **Transmitting Messages from an Extension to a Webview:** an extension transmits data to its webviews by calling the webview `.postMessage()` method. This approach enables the transfer of any JSON-serialisable data to the webview, where it is received via the standard message event.
- **Transmitting Messages from a Webview to an Extension:** webviews are capable of transmitting messages to their corresponding extension by utilising the `postMessage` function on a dedicated VS Code API object within the webview. This API object can be obtained by calling `acquireVsCodeApi` within the webview.

2.1.4 Eclipse Theia: The Hybrid Web Platform

Eclipse Theia is a framework specifically designed to provide an open, flexible platform for building custom, web-based IDEs. Unlike VS Code, which is a standalone product, Theia is a platform that allows adopters to build white-labelled tools using a modern web stack (TypeScript, Node.js) [21]. The term "Theia IDE" is intended as a ready-to-use IDE product built with the Theia platform.

The platform follows a client-server model where the UI runs in a browser (or Electron), and the backend manages data storage and computation. Theia can be viewed as a collection of extensions, each of which is packaged within an NPM package. Even the main cores are seen extensions (e.g. workspace, editor), providing the developer with a great way to customise the building IDE.

Theia's architecture is made of two main parts:

- **Frontend:** it runs in the browser or the Electron renderer. Manages UI, views, layout, menus, key-bindings, and the Monaco Editor.
- **Backend:** executes as a Node.js process, operating either locally in the Electron environment or remotely when used in a web browser. It is responsible for handling the file system, orchestrating language servers, coordinating debugging adapters, and supervising long-running operations.

The Frontend and Backend communicate thanks to the JSON-RPC protocol, and WebSocket is used in browser mode. The system connects components with dependency injection through InversifyJS.

Starting from the same codebase, a single Theia application can be packaged in many different ways. When used as a cloud IDE, Theia's frontend runs in the browser, and the backend runs on a dedicated server for each workspace. This setup works with containers and Kubernetes. The platform layer or a reverse proxy usually handles authentication, multi-tenancy, and workspace lifecycle management. Theia focusses on the delivery of IDE features. As a desktop application, it can be packaged using Electron, which bundles the Node.js backend and frontend together. Installers can be distributed and updates managed through standard Electron tools.

2.1.4.1 Extension Model

Theia doesn't just provide flexibility in terms of cross-platform development, but also from the extendability point of view. Extensions can be integrated in different ways, depending on the level of customisation needed.

- **Theia extensions** can obtain full access to the Theia API. They are installed at compile time and offer the best level of freedom for the IDE customisation. A Theia extension can communicate fully with the other IDE extensions. Considering that the core parts of Theia are also Theia extensions, it is possible to have direct communication with these components.

- **Theia plugins** work much like VS Code extensions, but allow access to some Theia-specific APIs and the frontend.
- **Headless plugins** are also similar to VS Code extensions. They are easy to develop and can be installed while Theia is running. These plugins only extend and interact with Theia backend services.
- **VS Code extensions** can be added through Theia's plugin host and Open VSX integration, where they get installed at runtime. This is a good way to reuse VS Code extensions, although this approach relies on the VS Code extension API and is so less powerful than the other approaches.

"Theia extensions are not restricted to a specific extension API, they can actually "access everything". In fact, the Theia core platform itself is built as Theia extensions." [46]. Unlike VS Code, which provides a ready-to-use product, Theia allows much deeper customisation and integration.

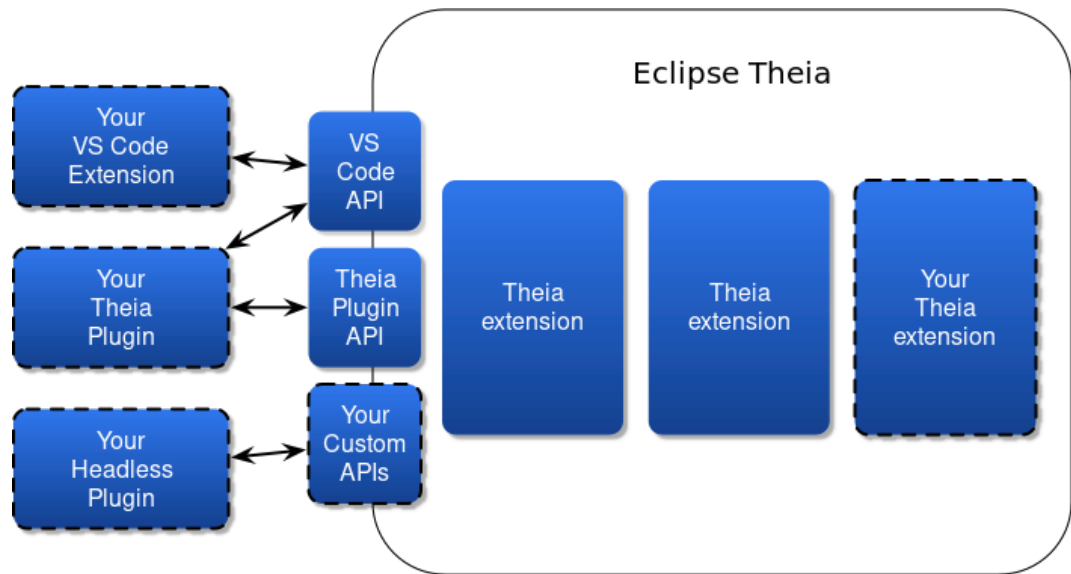


Figure 2.2: Theia extension types [47]

2.1.5 Special-Purpose Architectures (Equo SWT Evolve, Eclipse GLSP)

- **Equo SWT Evolve:** A modernisation bridge for legacy Eclipse RCP applications, allowing developers to execute existing SWT-based codebases within modern web browsers or as modernised desktop applications without requiring a full rewrite. Unlike the architectural shift required by platforms like Eclipse Che or GLSP, Equo SWT Evolve utilises a custom implementation of the SWT library that targets web technologies, effectively decoupling the "OSGi Monolith" from its native operating system dependencies. This enables a "lift-and-shift" migration path, preserving business logic while adopting a cloud-ready delivery model [48].

- **Eclipse GLSP:** The Graphical Language Server Platform transfers the Language Server Protocol approach to graphical editors. It centralises diagram-related logic in an independent server component, while client applications render the diagrams with technologies such as SVG and Sprotty, thereby supporting the transition of sophisticated modelling environments to cloud-based setups.

2.2 Web Frontend Frameworks

Frontend development is full of languages and frameworks to choose from. Considering the company's needs, it is also essential to evaluate the well-known programming languages and frameworks within the industry. An embedded company has a different approach to web development and also needs various types of constraints and distinct goals. Starting from Java and arriving at a more modern Rust, this section will detail the pros and cons of these languages within the thesis context of UI web development.

2.2.1 Java on the Frontend: Architectural Friction and Performance Overheads

Java is the programming language used for SWT and Eclipse RCP applications; the possibility of keeping it for the redesign of legacy applications is considerable. In addition to RAP, other new technologies have been developed. That can keep the codebase in Java and modernise the extensions at the same time, allowing them to work on the web.

Java is best known for backend services and enterprise middleware, but it also plays a role in some areas of frontend development. Java's first steps in browser-based frontend development used applets, which relied on browser plugins to run Java bytecode. When the web started to switch to HTML/CSS/JavaScript, web development changed its approach.

Since modern browsers do not run Java bytecode directly, the modern Java environment provides two main approaches to frontend development: server-driven UI frameworks that handle the user interface on the server and send back HTML and JavaScript code, and toolchains that compile Java code into JavaScript or WebAssembly for the client-side.

2.2.1.1 Server-driven UI

With server-driven UI, the programming model written in Java runs on the server. When users interact with the page, their actions are sent back to the server, which updates the UI state and returns only the changes. This approach offers strong type safety and tight backend integration, but it uses more server resources per user and can introduce extra delay for each interaction.

An example of a framework is Vaadin. "Vaadin includes a set of Web Components, a Java web framework, and a set of tools that enable developers to implement modern web graphical user interfaces (GUI) using the Java programming language only (instead of HTML and JavaScript), TypeScript only, or a combination of both." [49]. This type of technology can cause substantial network latency, making it unsuitable for developing IDE extensions.

2.2.1.2 Transpiling

The transpilation or compilation of client-side Java involves converting Java or JVM bytecode into JavaScript or WebAssembly. The app then runs directly in the browser, which enhances responsiveness and enables offline use. However, this method can lead to larger file sizes and makes it harder to work with native browser features.

WebAssembly is a great resource that will be highlighted in the following sections. But Java is not a good match for different reasons:

- **Java Bytecode and JVM-Centric Design:** Java is designed to run on the Java Virtual Machine (JVM), which provides features like garbage collection, Just-In-Time (JIT) compilation, and extensive runtime APIs. These are deeply integrated into Java's ecosystem and are not natively supported by WebAssembly.
- **Porting the JVM:** Running Java code as WebAssembly generally requires porting a JVM—or at least large parts of it—to WebAssembly, which adds extra overhead. Tools such as TeaVM and JWebAssembly try to compile Java into WebAssembly, but they depend on custom runtime environments that can be both limited in capability and unnecessarily large.
- **Server-Side Rendering:** Considering a possible migration of RAP to WebAssembly, RWT is a part of the Eclipse RAP (Remote Application Platform) framework, which uses a server-centric approach to render UIs. The UI logic runs on the server and the client (browser) only renders the frontend using JavaScript or HTML. This inherently relies on a live server-client interaction.
- **WebAssembly's focus:** WebAssembly is designed for running high-performance code directly in the browser, without relying on a backend server in many cases. This makes WebAssembly a poor fit for frameworks like RWT or many other Java frameworks that depend on server-side rendering and communication.

2.2.2 JavaScript and React: The Industry Standard and its Constraints

JavaScript (ECMAScript) is the native language of the web, and React is currently the dominant library for building user interfaces. For a modernisation project, React offers the highest developer velocity and the most extensive ecosystem of third-party libraries.

- **The Virtual DOM and Reconciliation:** React utilise a Virtual DOM to minimise direct manipulation of the browser's Document Object Model (DOM), which is computationally expensive. Although efficient for standard enterprise applications, the "reconciliation" process—where React compares the virtual and real DOM—can become a bottleneck when visualising high-frequency data from embedded traces.

- **The Single-Threaded Bottleneck:** JavaScript is single-threaded by nature. In the context of an engineering tool, heavy computational tasks (such as parsing large binary memory dumps) run on the same thread as the user interface. This can lead to "jank" or unresponsive interfaces during heavy processing, violating the 300 ms responsiveness threshold.
- **Garbage Collection (GC) Overhead:** Like Java, JavaScript relies on non-deterministic garbage collection. For tools requiring consistent low latency, the unpredictable pauses of the GC of the JavaScript engine can disrupt the "near-native" feel required for real-time signal visualisation.

While React is a vast improvement over Java-based server-side rendering for general-purpose UI, its performance "ceiling" necessitates the exploration of lower-level technologies for the core logic of embedded system tooling.

2.2.3 Rust + WebAssembly (Wasm): The Native-Speed Runtime

Rust is a systems programming language that provides the performance of C/C++ but with guaranteed memory safety. This is achieved through a unique "ownership" model enforced at compile-time, eliminating the need for a Garbage Collector (GC) [50]. Rust offers different functionalities that perfectly match the need for responsive feel, such as:

- **Near-Native Efficiency:** In the context of embedded system tooling, Rust is ideal because it lacks the "stop-the-world" pauses of GC-based languages (like Java or JavaScript). This ensures that heavy background tasks, such as protocol decoding or large-scale file parsing, do not freeze the UI [51].
- **Wasm Synergy:** Rust has become the premier language for WebAssembly development due to its minimal runtime and excellent toolchain support (wasm-pack) [50].

WebAssembly (Wasm) is a low-level binary instruction format designed as a portable compilation target for high-level languages like C++ and Rust. Unlike JavaScript, which is dynamically typed and requires Just-In-Time (JIT) compilation, Wasm is designed to be parsed and executed at near-native speed by the browser's engine [52].

- **Performance Characteristics:** Wasm bypasses the heavy parsing and "warm-up" phases of JavaScript. Its stack-based virtual machine allows for predictable execution times, which is critical for real-time data visualisation in embedded tooling.
- **Role in Modernisation:** It allows the "Core Logic" of a legacy Eclipse RCP application (if rewritten or bridged) to run in a sandboxed environment without the performance overhead traditionally associated with web technologies.

2.2.3.1 The Yew Framework: Component-Based Rust for the Frontend

Yew is a modern Rust framework used for creating multi-threaded frontend web applications. It uses a macro-based system (similar to React's JSX) to build UIs that are compiled entirely into Wasm.

- **Achieving Smooth UI:** While JavaScript toolkits regularly struggle with high-frequency DOM updates, Yew minimises the overhead of the JavaScript-to-Wasm bridge. By keeping the application state and logic entirely within the Wasm memory space, it reduces the cost of "serialisation/deserialisation".
- **Architectural Fit:** For and Eclipse RCP developer, Yew offers a familiar component-based architecture but provides the strict type of safety and performance required for low-level system views [23].

2.2.3.2 Slint: Compiled user interface for High-Performance Tooling

Slint (formerly SixtyFPS) is a modern UI toolkit designed specifically for memory-constrained and performance-critical applications. Unlike traditional web frameworks that rely on a runtime interpreter or a complex Virtual DOM reconciliation process, Slint uses a specialised UI description language that is compiled directly into highly optimised machine code or WebAssembly [53].

- **Ahead-of-Time (AOT) Compilation:** Slint's primary advantage for embedded systems tooling is its compilation model. The UI layout, styling, and property bindings are analysed and optimised at compile-time. This results in a minimal runtime footprint and near-instantaneous UI initialisation [53].
- **Near-Native Graphics:** Slint can render via multiple backends, including Skia or FEMTOVG. When targeted at WebAssembly, it can utilise **WebGL** or **WebGPU**, allowing for hardware-accelerated rendering of complex engineering views (such as real-time signal waveforms) that would typically cause significant "jank" in a standard JavaScript-based interface [54].
- **Memory Efficiency:** In the context of this thesis, Slint is a unique candidate because it does not require a heavy browser runtime to function. This makes it an ideal bridge where the same UI code can run as a native desktop application during development and as a Wasm module in a cloud-native environment [53].

Chapter 3

Methodology and Design

The transition of complex enterprise-grade software from legacy desktop environments to modern web architectures illustrates a paradigm shift in software engineering. This chapter establishes the methodological foundation for evaluating the strategic modernisation of proprietary Eclipse Rich Client Platform (RCP) applications. By applying a Design Science Research (DSR) lens, the framework delivers a coherent approach to balance the preservation of significant Java-based intellectual property with attaining genuine web efficiency and long-term native cloud suitability. The strategy detailed herein encompasses the research design philosophy, the systematic selection of case study artefacts, the definition of multi-dimensional architectural metrics, and the experimental deployment process comparing the actual Eclipse RCP against three distinct migration pathways: Eclipse Remote Application Platform (RAP), a standard web stack using React, and a WebAssembly solution using Slint. Following an initial comparative study, the research centres primarily on web-native pathways (React and Wasm) as the strategic target architectures for cloud-advanced maturity.

3.1 Scientific Foundation and Research Philosophy

The investigation of modernisation strategies for legacy toolsets requires a research paradigm that supports the development of original solutions to practical organisational challenges. Design Science Research (DSR) addresses the technical problems associated with re-architecting proprietary Eclipse RCP systems by emphasising the creation of prescriptive knowledge and purposeful artefacts that boost system functionality within a defined context.

This thesis applies DSR in accordance with the V-model, which systematically integrates scientific and technical domains. At the scientific level, the research studies existing knowledge regarding Java code reuse and web architecture performance. At the technical level, it introduces modernisation pathways. This approach makes certain that the resulting strategic framework constitutes a verifiable contribution to the software engineering knowledge base, rather than a routine design.

An essential component of the chosen methodology is the distinction between

discovery through action and discovery through design. The strategic framework developed here aims to generalise the class of problems defined by the technical obsolescence of desktop-first Java architectures.

3.2 Baseline Assessment: Technical Debt and Code Recyclability Analysis

Prior to evaluating migration pathways, establishing a quantitative baseline of the existing Eclipse RCP codebase is essential. This assessment fulfils two primary objectives:

1. Assessing the overall condition of the legacy system to evaluate its readiness for migration.
2. Identification of which components are suitable for reuse and which require complete reimplementatation.

3.2.1 Quantitative Models for Technical Debt

Managing the modernisation of proprietary Eclipse RCP applications requires a data-centric approach to quantifying technical debt. The industry-standard method for assessing debt is the Software Quality Assessment based on Lifecycle Expectations (SQALE) model, which delivers a coherent approach to estimating the remediation costs needed to bring a codebase into conformity with a defined quality model.

A key metric in this assessment is the Technical Debt Ratio (TDR), which measures the effort required to address existing debt relative to the total cost of developing the application from scratch:

$$TDR = \frac{\sum \text{Remediation Costs (minutes)}}{\text{Cost to Develop 1 LOC} \times \text{Total Lines of Code}}$$

Maintainability Rating	Technical Debt Ratio
A (Excellent)	$\leq 5\%$
B (Good)	6% to 10%
C (Fair)	11% to 20%
D (Poor)	21% to 50%
E (Critical)	$> 50\%$

Table 3.1: Maintainability Rating and Technical Debt Ratio

Within this approach, the default effort to implement a single line of code is set to 30 minutes. This metric is complemented by the Maintainability Index (MI), which incorporates Halstead measures and cyclomatic complexity. Quantitative analysis allows architects to identify which components of an existing Eclipse RCP tool are "migration-ready" and which require extensive refactoring.

3.2.2 Java Code Recyclability Assessment

Beyond measuring technical debt, the baseline assessment must determine the **Java Code Reuse Ratio**—the proportion of existing business logic that can be preserved during migration. This analysis distinguishes between:

- **UI-coupled code:** SWT/JFace components that are essentially tied to desktop presentation and require rewriting for web deployment
- **Business logic:** Domain-specific algorithms, validation rules, and data processing that can be extracted into "headless components" and reused across migration pathways
- **Infrastructure code:** OSGi bundles, Eclipse extension points, and platform-specific services that require varying degrees of architectural transformation depending on the chosen pathway

The reuse analysis implements static code analysis tools (SonarQube, SourceMeter) to calculate:

$$\text{Java Code Reuse Ratio} = \frac{\text{Reusable LOC (UI + Business Logic)}}{\text{Total Application LOC}} \times 100\%$$

This metric provides a realistic estimate of the migration effort and informs the strategic decision between preservation-focused (RAP) versus rewrite-focused (React/Wasm) pathways.

3.3 Taxonomy of Migration Strategies: Preservation vs. Web-Native Approaches

The analysis reviews three separate migration strategies, each defined by how they handle code reuse and architectural modernisation:

1. **UI Code Preservation (Eclipse RAP):** A server-side rendering solution intended to maximise reuse of existing Java/SWT application code by implementing an abstraction layer (RWT) that mimics the SWT API. This pathway aims at high code reuse (70-90%) by closely following the existing desktop architecture.
2. **Complete UI Rewrite using Modern Web Stack (React):** A full reconstruction of the presentation layer using industry-standard JavaScript/TypeScript frameworks. This approach delivers compatibility with contemporary cloud-native architecture practices while making use of the extensive React ecosystem and its large developer community. Communication with the retained Java backend logic is handled via REST APIs, strengthening a stateless client-side rendering (CSR) model.

3. **High-Performance UI with WebAssembly (Slint):** A Rust-based UI framework compiled to WebAssembly for near-native performance in the browser. This pathway targets compute-heavy applications that require consistent 60 FPS rendering and minimal execution overhead. Similarly to React, it consumes backend services via APIs but offers superior execution speed for CPU-bound operations.

3.3.1 Exclusion of Partial Modernisation Approaches

Additional technologies are excluded from the primary analysis, as they do not provide holistic migration frameworks:

Eclipse GLSP operates as a specialised protocol for diagramming editors, focusing on the internal state of specific graphical components rather than providing a mechanism to migrate the SWT/JFace standard application workbench, menu structures, or global state management.

Equo SWT Evolve offers benefits in terms of maximising UI code reuse via Chromium-based rendering. However, its core implementation is limited to single-user usage and essentially operates as a "web-enabled" desktop application. In contrast to client-server architectures, it depends on costly container orchestration (one instance per user) to simulate multi-user behaviour, and therefore does not satisfy the requirements for resource-effective server deployment.

Technology	Scope	Deployment Model	Research Role
Eclipse GLSP	Diagram/Graphical Editors	Web Components	Excluded: Component-specific, not holistic
Equo SWT Evolve	Full SWT UI	Single-User Browser/Desktop	Excluded: Single-user constraint
Eclipse RAP	Full Workbench UI	Multi-User Server-Side	Included initially, excluded after architectural analysis
React	Full UI Rewrite	Stateless Client-Side	Primary strategic pathway
WebAssembly (Slint)	Full UI Rewrite	Stateless Client-Side	Primary strategic pathway

Table 3.2: Migration Technology Scope and Research Inclusion Rationale

3.4 Case Study Selection and Implementation Scope

To validate the migration framework across several application characteristics, the research uses two complementary case studies:

3.4.1 Case Study 1: Standard Form-Based Application

Purpose: Evaluate general-purpose migration capability for typical business logic applications.

Characteristics:

- Intermediate UI complexity featuring typical form fields, input validation, and full CRUD functionality.
- Representative of the majority of Eclipse RCP enterprise applications.
- Minimal operations related to performance.

Evaluation Focus: This case study functions as the main reference point for comparing the three pathways, with an in-depth assessment of:

- Required development effort and proportions of code reuse.
- Maintainability indicators (TDR, code quality evaluations).
- Conventional web performance measures (TTI, INP, Lighthouse results).
- Characteristics of cloud-based deployment.

Rationale for parameter choice: Because React and Wasm share broadly similar architectural characteristics for typical business applications, this evaluation emphasises a targeted group of key migration decision criteria rather than an exhaustive feature-by-feature comparison:

- **Maintainability:** Quality of the codebase after migration (SonarCloud assessments).
- **Development velocity:** Duration required to reach full feature parity.
- **Performance baseline:** Core responsiveness indicators (INP target <200 ms).
- **Ecosystem interoperability:** Interoperability with existing tooling and overall developer experience.

Expected Outcome: Confirm that, for standard business applications, React and Wasm result in comparable performance with primary differentiation in ecosystem maturity rather than performance characteristics.

3.4.2 Case Study 2: Graphics-Intensive Application

Objective: Assess efficiency measures and identify the most suitable scenarios for deploying high-performance pipelines.

Characteristics:

- Intensive rendering workloads with continuous, real-time visual updates.
- Compute-intensive operations that must consistently maintain 60 FPS.
- Minimal business logic complexity (deliberately kept simple to prioritize rendering performance).

Evaluation Focus: This case study is designed to empirically confirm Slint’s performance benefits in the following areas:

- Consistency of frame rate under load (FPS profiling).
- Memory efficiency for graphical operations.
- CPU usage within the rendering pipeline.
- Binary footprint compared to JavaScript bundle overhead.

Strategic Purpose: Offers explicit criteria for determining under which conditions the performance characteristics of WebAssembly sufficiently warrant its comparatively higher initial development overhead and its more limited surrounding ecosystem relative to React.

3.4.3 Implementation Process

All implementations were developed with external technical support and adhered to a unified architectural strategy:

1. **Baseline Extraction:** Separation of business logic from user interface-dependent code in the existing Eclipse RCP applications.
2. **RAP Implementation:** Maximisation of code reuse by adapting SWT/JFace components to RWT with minimal source code modifications.
3. **React Implementation:** Complete redesign of the user interface using TypeScript, React hooks, and a contemporary component-based architecture.
4. **Wasm Implementation:** Development of a declarative user interface in Slint with a Rust-based backend, compiled into a WebAssembly binary.
5. **Standardised Testing:** Application of uniform testing protocols throughout all implementation variants to ensure comparability and internal validity.

3.5 Architectural Metrics for Cross-Platform Comparison

The framework defines numerical measures to compare the existing Eclipse RCP application against all three migration pathways, enabling data-driven strategic decisions.

3.5.1 Primary Evaluation Measures: Maintainability and Code Quality

Given that all migration pathways are required to produce maintainable, production-grade software artifacts, the principal basis for comparison is the quality of the code after migration. The following quantitative and qualitative metrics are employed:

- **Technical Debt Ratio (TDR):** An SQALE-based metric computed for each migration pathway's resulting implementation.
- **Maintainability Rating:** The letter-grade (A-E) assessment generated by SonarCloud, indicating the maintainability level of the analysed codebase.
- **Code Duplication:** The proportion of code blocks identified as duplicated within the codebase, reported as a percentage.
- **Cyclomatic Complexity:** The mean cyclomatic complexity per function or method, serving as an indicator of control-flow complexity.
- **Test Coverage:** The proportion of code statements or branches exercised by automated tests, expressed as a percentage.

3.5.2 Secondary Measurement Metrics: Performance and End-User Interaction

For the standard form-based web application, performance assessment is conducted with a focus on established web responsiveness criteria:

- **Interaction to Next Paint (INP):** A Core Web Vital metric that quantifies overall responsiveness through measuring the latency of users' interactions, with a target threshold of < 200 ms.
- **Time to Interactive (TTI):** The elapsed time from initial page load until the application becomes fully interactive, i.e., capable of reliably responding to user input without considerable delay.
- **Lighthouse Performance Score:** A composite metric derived from multiple performance indicators (e.g., loading speed, interactivity, visual stability), serving as an aggregate measure of client-side load performance.

For the graphics-intensive application, performance evaluation emphasises rendering performance:

- **Frames Per Second (FPS):** Consistency of the 60 FPS target under load.
- **Memory Footprint:** Utilisation of client resources.
- **CPU Utilisation:** Degree to which the processor is used during rendering tasks.

3.5.3 Cloud-Native Readiness Assessment

Modernisation should enable a progression towards a "Cloud-advanced" level of maturity, assessed based on:

- **Statelessness:** Capability to restart or replace instances without impacting user state.

- **Horizontal Scalability:** Ability to scale elastically without architectural limitations (such as sticky sessions).
- **Container Density:** Productive resource utilisation that supports high-density container deployments.
- **Ecosystem Alignment:** Effortless integration with contemporary CI/CD pipelines, observability solutions (e.g., OpenTelemetry), and cloud-native platforms (such as Kubernetes).

3.5.4 Comparative Study: Tools and Parameters

Tool Category	Tool/Method	Parameter Measured
Code Quality Analysis	SonarCloud	TDR, Maintainability Rating, Code Duplication, Complexity
Client Performance	Chrome DevTools Performance Panel	FPS, INP (form app only)
Load Performance	Google Lighthouse	Performance Score, TTI, Bundle/Binary Size
Graphics Performance	Chrome DevTools Rendering Panel	FPS consistency, frame drops (graphics app only)
Memory Analysis	Chrome Memory Profiler	Client-side JavaScript/Wasm memory footprint
Cloud-Native Fit	CNCF Cloud Maturity Model	Statelessness, Scalability, Container Density

Table 3.3: Comparative Study Tools and Parameters

3.6 Strategic Decision Framework

The procedure adopts explicit criteria for the phased evaluation and potential exclusion of Eclipse RAP following initial maintainability and architectural analysis.

3.6.1 RAP Evaluation Criteria

Eclipse RAP is evaluated along three key dimensions:

1. **Maintainability:** Does RAP attain an "A" rating on SonarCloud, signifying production-grade code quality?
2. **Architectural Soundness:** Is RAP's server-side, stateful architecture compatible with cloud-native requirements for stateless, horizontally scalable services?
3. **Development Experience:** To what extent do RAP's IDE integration mechanisms and associated tooling conform to state-of-the-art web development practices, including the npm-based ecosystem, TypeScript language support, and the use of modern component libraries?

3.6.2 React vs. Wasm Selection Framework

Following RAP evaluation, the framework delivers advice for choosing between React and Wasm based on application characteristics:

Decision Factor	React Recommended	WebAssembly (Slint) Recommended
Performance Requirements	Standard business applications with moderate interactivity	CPU-intensive applications requiring consistent 60 FPS (e.g., CAD, simulations, real-time visualisation)
UI Complexity	Rich interactive interfaces with complex state management	High-performance graphical rendering with minimal DOM manipulation
Developer Availability	Large pool of JavaScript/-TypeScript developers readily available	Willingness to invest in Rust/Wasm training or hire specialised talent
Ecosystem Needs	Extensive third-party library requirements (charting, forms, authentication)	Performance-critical core with limited external dependencies
Time to Market	Rapid development with mature tooling and component libraries	Longer initial development acceptable for long-term performance gains

Table 3.4: Strategic Pathway Selection Criteria

3.7 Research Execution Summary

This methodology culminates in an organised evaluation process:

3.7.1 Phase 1: Universal Maintainability Assessment

- Conduct SonarCloud analysis on all three migration implementations (RAP, React, Wasm).
- Establish that all pathways attain an “A” rating for maintainability, thereby evidencing their capability for production deployment.
- Systematically identify architectural restrictions in RAP (including the stateful server model and IDE integration frictions) that constrain its suitability for cloud-native environments.

3.7.2 Phase 2: Standard Application Comparison (React vs. Wasm)

- Evaluate form-based application with focused parameter set (maintainability, basic performance, development effort).
- Show functional parity between React and Wasm for standard business logic.
- Establish React as the default recommendation based on the maturity of the ecosystem and the availability of the developers.

3.7.3 Phase 3: Validation with Graphics-Intensive Applications

- Deploy a minimal yet representative graphics-intensive application to isolate and quantify rendering performance characteristics.
- Empirically validate Slint’s ability to sustain a rendering rate of 60 FPS, while appraising its comparative advantages in resource utilisation (e.g., CPU, memory, and GPU usage).
- Delineate precise application scenarios and operating conditions under which the performance characteristics of WebAssembly (Wasm) justify potential trade-offs associated with the surrounding ecosystem.

Solution	Code Reuse	Cloud-Native Fit	Primary Use Case
Eclipse RCP (Base-line)	100%	Legacy (On-Prem)	Desktop Enterprise Tools
Eclipse RAP	70-90% (UI+Logic)	Cloud-Enabled (Stateful)	Transitional/Interim Solution
React	Backend Logic Only	Excellent (Stateless)	General Business Applications
WebAssembly (Slint)	Backend Logic Only	Excellent (Stateless)	Graphics-Intensive Applications

Table 3.5: Strategic Pathway Comparison Summary

This well-structured methodology delivers the evidential basis for a strategic modernisation framework that acknowledges the pragmatic appeal of code preservation (RAP), while ultimately directing organisations toward web-native architectures (React/Wasm) that correspond with long-term cloud scalability and ecosystem sustainability aims.

Chapter 4

Implementation of Modernisation Pathways

The transition of legacy proprietary software architectures from desktop-centric frameworks to cloud-ready web platforms represents one of the most complex challenges within modern enterprise software engineering. Within the automotive industry, where development toolchains for embedded systems have previously relied on the Eclipse Rich Client Platform (RCP), the need for modernisation is motivated by shifting user expectations, the increase of remote collaborative workflows, and the extended sustainability risks associated with ageing technology stacks. This empirical case study, which serves as the core evaluative component of the proposed strategic framework, documents the implementation and results of four distinct modernisation strategies.

4.1 Custom SWT Widgets

Infineon Technologies Eclipse RCP applications are built using custom SWT widgets, which are then packaged and distributed within a company's internal library. To understand how they work, it is essential to show the class details in terms of implementation and dependencies.

4.2 Implementation Strategy 1: Eclipse RAP

The first implementation path explores the Eclipse Remote Application Platform (RAP) as a bridge to modernise legacy Eclipse RCP applications with minimal code disruption. This minimal-change approach focuses on "Single Sourcing", a pattern in which the same codebase can target both desktop and web runtimes.

4.2.1 Architectural Mapping

Moving from Eclipse RCP to RAP mainly involves swapping dependencies. The RAP Widget Toolkit (RWT) shares most of the classes and methods of SWT, making easy mapping possible. To facilitate the management and deployment of RAP applications, Spring Boot can be used to build a fully functional stand-alone web application.

The term "Spring RAP" will be used in the thesis context to describe an Eclipse RCP application ported to a web environment using Eclipse RAP and Spring Boot. Support for the Apache Maven build automation tool enables dependency management and allows adding the necessary application dependencies to the project's "pom.xml" file.

All Eclipse platform dependencies that are needed to make the application work are the following:

- org.eclipse.rap.rwt
- org.eclipse.core.resources
- org.eclipse.equinox.common
- org-apache-commons-io
- org.eclipse.ui

4.2.2 Entry Point Configuration and Lifecycle

Unlike the native RCP workbench, Spring RAP requires an explicit bridge between the Spring context and the RWT lifecycle. This is achieved through the following:

- **A class (or more) extending `AbstractEntryPoint`.** In this, the "createContent" method is annotated with "@PostConstruct", which guarantees that it is called exactly once, immediately after Spring Beans are initialised. This will be the application Entry point that contains the Eclipse RAP application.
- **A class implementing "`ApplicationConfiguration`".** This can configure all RAP Entry Points, application properties, and operation modes.
- **A class implementing "`ServletContextInitializer`".** In the "onStartup" method of this class, the RWT `ApplicationRunner` is used to start the application.

- **A class containing a `ServletRegistrationBean`**, used to register a new custom RWT Servlet. As shown in the snippet of code below.

```

1  @Bean
2  public ServletRegistrationBean<RWTServlet> rwtServlet() {
3      var bean = new ServletRegistrationBean<>(new RWTServlet());
4      bean.addUrlMappings("/rap");
5      return bean;
6  }

```

4.2.3 The Headless State API: REST Controller Integration

To allow external tools (like the VS Code extension) to interact with the application state, a REST Controller was implemented. This replicates the inter-process communication (IPC) of a desktop environment in a web-native way.

To be compatible with this strategy, the RAP Entry Point requires some modifications. Since the application will be run on the machine's localhost, only one copy of the Entry Point is required. This follows the Singleton pattern, permitting a single instance of the Entry Point. In addition, some new static attributes must be added to the class. Making them static will enable the retrieval of their values inside the REST Controller:

- **private static `Display parent`**: assumes the value of "`Display.getDefault`" called inside the "`createContent()`" method of the Entry Point. This provides access to the UI thread from within the REST Controller.
- **private static `Map<String, IADS2WidgetWithValue<?> > widgets`**: a map containing all the application widgets. Each key is intended as the "`widgetName`" of the widget, while the value is the widget itself.

Inside the REST Controller, the "`@GetMapping`" and "`@ResponseBody`" annotations can be used to map the widget values to a single endpoint, for example, "`https://localhost:<PORT>/rap/api`".

4.2.4 Technical Challenges and Solutions

4.2.4.1 Key constants

Some custom SWT widgets use character fields (`SWT.ESC`, `SWT.DEL`, `SWT.TAB`, `SWT.BS`, `SWT.CR`), which are not compatible with RAP. As a solution, all the needed character fields can be replaced with the corresponding `keyCode`, as shown in the figure.

public static final int	MouseDoubleClick	8
public static final int	MouseDown	3
public static final int	MouseEnter	6
public static final int	MouseExit	7
public static final int	MouseMove	5
public static final int	MouseUp	4
public static final int	MouseWheel	37
public static final int	Move	10

Figure 4.3: Character fields and corresponding keyCode

4.2.4.2 Network latency

Some of the custom classes use specific events that require a rapid response. This can be easily achieved with SWT, as it utilises the native OS. On the other hand, RWT is made for the web and cannot rely on this type of resource.

In the case of the "GCombo" widget, a menu detect listener is initialised to check whether the mouse wheel is used when the combo menu is opened. In that case, it sets the MenuDetectEvent "doit" attribute to "false", a flag indicating whether the operation should be allowed. Setting this field to "false" will cancel the operation.

For a mouse wheel check, a minor delay is needed. It becomes very difficult to handle these types of events on the server side due to network latency. This problem can be solved with RWT Scripting, which can handle certain events directly on the client side without requiring network communication. This type of script must be written in JavaScript using an SWT-like API and passed to a ClientListener that is added to the widget. In this way, the mouse wheel event will be handled on the client-side. Fig. 4.4 shows how the ClientListener replaces the old code.

+/-	Line #	Content
-	1	<code>this.addListener(SWT.MouseVerticalWheel, (e) -> e.doit = false);</code>
+	1	<code>ClientListener listener = new ClientListener(</code>
+	2	<code> "function handleEvent(event){ " +</code>
+	3	<code> " if (typeof event.preventDefault === 'function') {" +</code>
+	4	<code> " event.preventDefault();" +</code>
+	5	<code> " } else {" +</code>
+	6	<code> " event.doit = false;" +</code>
+	7	<code> " }" +</code>
+	8	<code> "}"</code>
+	9	<code>);</code>
+	10	
+	11	<code>this.addListener(SWT.MouseWheel, listener);</code>
+	12	<code>[empty line]</code>

Figure 4.4: Code difference between SWT and RWT Mouse Wheel listener implementation

4.2.4.3 Link handling

In SWT, the class "org.eclipse.swt.program.Program" is used to open a link in a web browser. This is strictly dependent on the OS, as a new process is launched. This cannot be reasonably implemented with RWT, which instead provides a way to open any URL without leaving the current page or session via the URL launcher. Fig. 4.5 shows how the UrlLauncher is initialised.

+/-	Line #	Content
-	2	<code>Program.launch(linkURL);</code>
+	2	<code>UrlLauncher launcher = RWT.getClient().getService(UrlLauncher.class);</code>
+	3	<code>launcher.openURL(linkURL);</code>
+	4	<code>[empty line]</code>

Figure 4.5: Code difference between SWT and RWT link handling

4.2.5 Thread Synchronisation

Given how threads are handled in Eclipse RAP, to make the REST Controller work, it is necessary to synchronise threads. This can be achieved by calling the "syncExec()" function on the "Display parent" static attribute of the Entry Point. This will allow the REST Controller to request access to the UI thread for some of its information.

To signal to the UI thread that the work of the "syncExec()" function is completed,

a new static attribute is added to the Entry Point:

- **private static Boolean done:** this attribute gets initialised to false and then set to true by the REST Controller when there is no more need to access UI thread resources.

```

1  @GetMapping("/rap/api")
2  @ResponseBody
3  public synchronized String getWidgetsData() {
4      final AtomicReference<String> result = new
AtomicReference<String>();
5      Display parent = RAPEntryPoint.getParent();
6      if (parent != null && !parent.isDisposed()) {
7          parent.syncExec(() -> {
8              Map<String, IADS2WidgetWithValue<?>> widgets =
RAPEntryPoint.getInstance().getWidgets();
9              Boolean first = true;
10             String res = "{";
11             for (String id : widgets.keySet()) {
12                 if (widgets.get(id) instanceof GCombo) {
13                     IADS2WidgetWithValue<String> widget =
(IADS2WidgetWithValue<String>) widgets.get(id);
14                     widget.setValue((((GCombo) widget).getText()));
15                 } else if (widgets.get(id) instanceof GCheck) {
16                     IADS2WidgetWithValue<Boolean> widget =
(IADS2WidgetWithValue<Boolean>) widgets.get(id);
17                     widget.setValue((((GCheck)
widget).getSelection()));
18                 } else if (widgets.get(id) instanceof GFloat) {
19                     IADS2WidgetWithValue<String> widget =
(IADS2WidgetWithValue<String>) widgets.get(id);
20                     if((((GFloat) widget).isEnabled())
21                     widget.setValue((((GFloat)
widget).getText()));
22                 }
23                 if (first) {
24                     res += "\n\t\"" + id + "\": \"" +
widgets.get(id).getValue() + "\"";
25                     first = false;
26                 }
27                 else
28                     res += ",\n\t\"" + id + "\": \"" +
widgets.get(id).getValue() + "\"";
29                 }
30                 res += "\n}";
31                 result.set(res);
32                 RAPEntryPoint.setDone(true);
33             });
34         }
35         return result.get();
36     }

```

4.2.6 SRAPA: Spring RAP Automation

Automation tools are becoming increasingly common in the development field, easing developer work and providing a starting-point template. VS Code and Theia are examples of platforms that have created generators for their own projects, thanks to the Yeoman tool.

Spring RAP Automation (SRAPA) is a tool built within the context of this thesis to enable the creation of a Spring RAP application from an Eclipse RCP application.

Starting from the Java class that contains the SWT implementation of an application, the tool automatically builds a working Spring RAP application by moving the SWT content inside an RAP Entry Point. All configuration is automatically handled, allowing developers to focus on the most critical work.



Figure 4.6: Spring RAP Automation (SRAPA)

4.3 Implementation Strategy 2: React

This implementation pathway represents a "Clean Break" strategy. In contrast to the Transpositional Pathway (Eclipse RAP), which attempts to preserve the server-side OSGi lifecycle, this approach treats the legacy Eclipse RCP code as a functional specification. The UI layer is reimplemented entirely within the React ecosystem, maintaining business logic parity through a rigid architectural mapping rather than code reuse.

The choice of React for the modernisation of enterprise Rich Client Platform (RCP) tools is motivated by the framework's mature and extensive ecosystem, the widespread availability of specialised frontend engineering expertise, and its well-known ability to support complex, state-intensive workbench interfaces. Accordingly, the central objective of the implementation is the systematic reconstruction of the flexible, tab-based docking mechanism that characterises the Eclipse user interface, and its adaptation to a contemporary, web-based runtime environment.

4.3.1 Architectural Mapping: The "Workbench" to "DOM" Translation

The primary engineering challenge in migrating from Eclipse RCP to React is not the linguistic translation (Java to TypeScript), but the paradigmatic shift from an imperative, object-oriented UI toolkit (SWT/JFace) to a declarative, state-driven DOM library. To ensure that the migrated application retains the "Power User" capabilities of the existing tool—specifically docking views, resizing panels, and managing multiple editor tabs—standard web CSS layouts (Flexbox/Grid) were augmented with a specialised Docking Layout Engine.

The following table details the translation of core Eclipse RCP concepts as their web-native counterparts:

Eclipse RCP Concept	Web Implementation	Rationale
Workbench Window	FlexLayout.js / Golden-Layout	Provides "Docking Manager" capabilities, enabling drag-and-drop tabs and resizable split-panes, mimicking the <code>MPartStack</code> behavior of Eclipse e4.
Perspective	Layout Model (JSON)	The Eclipse <code>perspective.xml</code> is serialized into a JSON state object that defines the initial grid structure. This is functionally equivalent to the Eclipse e4 Application Model (<code>fragment.e4xmi</code>).

Eclipse RCP Concept	Web Implementation	Rationale
ViewPart / Editor-Part	React Functional Components	Each RCP View is encapsulated as an isolated React component. Lifecycle methods (<code>createPartControl</code> , <code>dispose</code>) are mapped to <code>useEffect</code> hooks.
Selection Service (ISelectionService)	React Context + Hooks	A global <code>SelectionContext</code> replaces the OSGi selection service. To prevent performance bottlenecks, a "Selector Pattern" is used to ensure only components interested in specific data types re-render.
JFace Databinding	React Hook Form / Zod	The observable value pattern of JFace is replaced by controlled inputs and schema-based validation, decoupling the UI from the underlying data model.
Command Framework (IHandler)	Custom Hooks + Command Registry	Commands are registered in a global Map and invoked via keyboard shortcuts (<code>react-hotkeys-hook</code>) or toolbar buttons. The command pattern is preserved while adapting to event-driven web paradigms.
Extension Points	Plugin Architecture (Dynamic Imports)	Third-party extensions can be loaded via dynamic <code>import()</code> statements, with a manifest file defining contribution points. This replicates the extensibility model without OSGi overhead.
Preferences (IPreferenceStore)	LocalStorage + Context	User preferences are persisted to browser LocalStorage and exposed via a <code>PreferencesContext</code> for reactive updates across the application.

Table 4.1: Component Architectural Mapping with React

4.3.2 Workbench Replication and Layout Management

Eclipse RCP's defining feature is the workbench. In a React-native implementation, this functionality is replicated using a layout manager that arranges components into tab sets. The implementation entails defining a JSON model that represents the workbench's hierarchy. A factory function maps node types to specific React

components using Code Splitting (lazy loading) to replicate the "Lazy Activation" of Eclipse plugins—making sure that heavy components are only loaded when requested by the user.

```
1 {
2   "global": {
3     "tabSetEnableMaximize": true,
4     "tabEnableClose": true,
5     "tabSetEnableDrag": true
6   },
7   "layout": {
8     "type": "row",
9     "children": [
10      {
11        "type": "tabset",
12        "weight": 20,
13        "children": [
14          {
15            "type": "tab",
16            "name": "Project Explorer",
17            "component": "com.example.rcp.views.ProjectExplorer"
18          }
19        ]
20      },
21      {
22        "type": "tabset",
23        "weight": 80,
24        "children": [
25          {
26            "type": "tab",
27            "name": "Form Editor",
28            "component": "com.example.rcp.editors.FormEditor",
29            "config": { "documentId": "doc_123" }
30          }
31        ]
32      }
33    ]
34  }
35 }
```

Listing 4.1: Workbench Layout Model (JSON Serialisation)

This JSON structure replicates the Eclipse e4 Application Model, providing a declarative definition of the initial workbench state. The layout engine interprets this model at runtime, instantiating the appropriate React components and arranging them according to the specified hierarchy.

```

1  import React, { Suspense } from 'react';
2  import { ErrorBoundary } from 'react-error-boundary';
3
4  // Lazy load components to mimic OSGi "Lazy Activation"
5  const ProjectExplorer = React.lazy(() =>
6    import('./views/ProjectExplorer'));
7
8  const FormEditor = React.lazy(() => import('./editors/FormEditor'));
9
10
11  const FallbackView = () => <div>View Loading...</div>;
12  const ErrorFallback = () => <div>View Failed to Load</div>;
13
14  // The "View Registry" replaces the Eclipse Extension Point mechanism
15  export const ViewFactory = (node) => {
16    const componentId = node.getComponent(); // e.g.,
17    "com.example.view"
18    const config = node.getConfig();
19    let Component;
20
21    switch (componentId) {
22      case "com.example.rcp.views.ProjectExplorer":
23        Component = ProjectExplorer;
24        break;
25      case "com.example.rcp.editors.FormEditor":
26        Component = FormEditor;
27        break;
28      default:
29        return <div>Unknown View: {componentId}</div>;
30    }
31
32    // Wrap in ErrorBoundary and Suspense for robustness
33    return (
34      <ErrorBoundary fallback={<ErrorFallback />>
35        <Suspense fallback={<FallbackView />>
36          <Component session={config} />
37        </Suspense>
38      </ErrorBoundary>
39    );
40  };

```

Listing 4.2: The View Registry Factory Pattern

4.3.3 Enterprise-Scale State Management

The transition from the OSGi service registry to web state management requires the adoption of functional reactive patterns. In the legacy OSGi environment, services were discovered and injected dynamically, maintaining a decoupled, yet strictly typed, lifecycle. In the modern web stack, this decoupling is accomplished through a centralised "Single Source of Truth." For enterprise-scale tools, the Redux Toolkit (RTK) remains the industry standard for managing the "View State"—the metadata, which describes the current state of the UI components, such as side-panel expansion,

active tabs, and modal visibility.

However, a common complexity in large React-based redesigns is "prop drilling," where the state is passed through multiple layers of layout containers that do not actually need the data, acting merely as couriers. To reduce this, the implementation uses the React Context API to share state across component boundaries without polluting the intermediate layers.

4.3.3.1 Context Optimisation: The Selector Pattern

To prevent unnecessary re-renders when using Context for global state, the implementation uses a selector-based subscription model. This pattern makes certain that components only re-render when the specific slice of state they depend on changes, rather than on every context update.

```

1  import { createContext, useContext, useMemo } from 'react';
2
3  interface SelectionState {
4      activeFile: FileDescriptor | null;
5      activeView: string | null;
6      selectedItems: string[];
7      cursorPosition: { x: number; y: number };
8  }
9
10 // Context holds the full selection state
11 const SelectionContext = createContext<SelectionState | null>(null);
12
13 // Custom hook that accepts a selector function
14 function useSelection<T>(selector: (state: SelectionState) => T): T {
15     const fullState = useContext(SelectionContext);
16     if (!fullState) {
17         throw new Error('useSelection must be used within
SelectionProvider');
18     }
19     // Only re-render if the selected slice changes
20     return useMemo(() => selector(fullState), [fullState, selector]);
21 }
22
23 // Usage: Component only re-renders when 'activeFile' changes
24 function FileProperties() {
25     const activeFile = useSelection(state => state.activeFile);
26     // This component ignores changes to state.activeView,
27     // state.selectedItems, state.cursorPosition, etc.
28
29     if (!activeFile) return <div>No file selected</div>;
30
31     return (
32         <div><h3>{activeFile.name}</h3>
33         <p>Size: {activeFile.size} bytes</p>
34         </div>
35     );
36 }

```

Listing 4.3: Optimised Selection Context with Selectors

This selector pattern is critical for retaining performance in complex workbench layouts where multiple components subscribe to different aspects of the global state.

4.3.4 Remote Data Synchronisation

Synchronising local data with backend services is handled by TanStack Query (formerly React Query). This tool addresses the "Remote State" problem by automating API fetches, caching data, and handling invalidation, replacing the manual boilerplate often found in raw Redux implementations. This is particularly relevant when migrating RCP applications that rely on persistent database connections; TanStack Query enables "optimistic updates", maintaining UI responsiveness even in high-latency cloud environments by updating the UI before the server response is confirmed.

4.3.5 Bidirectional Integration: The Headless State API

To allow external services to query and manipulate the form state—replicating the inter-process communication (IPC) often found in Eclipse RCP plugins—the React implementation utilises the Service Provider Pattern.

Instead of the form state being trapped within a component's local scope, it is exposed via a custom Integration Controller. This controller operates as a bridge between the internal React state (managed by Hook Form) and an external-facing REST or WebSocket API.

- **Inbound Requests:** External components may retrieve the current field values via GET operations directed to either a local listener or a shared state bus.
- **Outbound Synchronisation:** Upon each state transition, the application may optionally submit modified ("dirty") fields using POST requests to an intermediary middleware service, thereby preserving consistency between the external system and the user's input.

While TanStack Query manages the persistence of data to remote cloud services, the Headless State API makes certain that this same data is accessible to external processes running locally, effectively replicating the integrated ecosystem of a native RCP environment.

4.3.6 Technical Challenges and Solutions

4.3.6.1 The "Blocking Dialogue" Problem

In Eclipse SWT, invoking `MessageDialog.openConfirm()` blocks the execution of the calling method while the UI event loop continues to process events. JavaScript,

being single-threaded and non-blocking, does not support this "pause" mechanic; awaiting a user interaction without yielding the thread would freeze the browser.

The proposed approach is implemented as a promisified modal service. Rather than relying on synchronous, blocking execution, the application utilises the JavaScript event loop. Control flow is yielded to the browser's task queue via the `async/await` syntax, thereby transferring the responsibility for state preservation from the call stack to a JavaScript closure.

```
1 // Define the modal state interface
2 interface ModalState {
3     isOpen: boolean;
4     title: string;
5     message: string;
6     onConfirm: () => void;
7     onCancel: () => void;
8 }
9
10 // Define the service interface
11 interface IModalService {
12     confirm(title: string, message: string): Promise<boolean>;
13 }
14
15 // Implementation using a React Context or simple singleton
16 class ModalService implements IModalService {
17     private setModalState: (state: ModalState | null) => void;
18
19     constructor(setModalStateFn: (state: ModalState | null) => void) {
20         this.setModalState = setModalStateFn;
21     }
22
23     confirm(title: string, message: string): Promise<boolean> {
24         return new Promise((resolve) => {
25             // Trigger the UI (e.g., set React state to show the
26             // modal)
27             this.setModalState({
28                 isOpen: true,
29                 title,
30                 message,
31                 // The UI button clicks will trigger these callbacks
32                 onConfirm: () => {
33                     this.setModalState(null); // Close modal
34                     resolve(true); // Resume execution with 'true'
35                 },
36                 onCancel: () => {
37                     this.setModalState(null); // Close modal
38                     resolve(false); // Resume execution with 'false'
39                 }
40             });
41         });
42     }
43 }
```

```
44 // Usage in a migrated Handler
45 async function handleDelete() {
46     // Execution yields here; the UI stays responsive
47     const shouldDelete = await modalService.confirm(
48         "Confirm",
49         "Delete file?"
50     );
51
52     if (shouldDelete) {
53         await fileService.delete(selectedFile);
54     }
55 }
```

Listing 4.4: Promisified Modal Service Implementation

4.3.6.2 The Broadcast Problem

While the Context API solves the visibility issue, it creates a potential performance bottleneck. By default, when the value of a Context provider changes, every component that consumes that context is forced to re-render. In a complex "Workbench" UI—where a single state change might be as small as a cursor coordinate or a checkbox toggle—triggering a re-render of the entire layout tree is computationally expensive. In a large enterprise application, the React "Reconciliation" process (comparing the virtual DOM to the real DOM) can take several milliseconds. If this happens on every keystroke or mouse movement due to a top-level Context update, the UI will feel "heavy" or "laggy" to the user.

To sustain high performance of the workbench, the implementation must employ three primary memoization strategies:

- **React.memo:** This higher-order component inhibits the re-rendering of a functional component when its properties remain unchanged. By encapsulating computationally expensive layout containers within `memo`, it is possible to effectively short-circuit the render phase and thereby reduce unnecessary rendering overhead.
- **useMemo:** Used within components to cache the result of expensive calculations. For example, if the UI needs to filter a list of 1,000 OSGi bundles based on a search term, `useMemo` ensures that filtering occurs only when the search term or the bundle list changes.
- **useCallback:** This hook is especially important when you pass functions down via the Context API. Because functions are objects in JavaScript, a new instance is created on every render. `useCallback` retains the same function reference across renders, so child components don't mistakenly detect a "changed prop" and rerender unnecessarily.

4.4 Implementation Strategy 3: WebAssembly with Slint

This implementation pathway represents the "Performance-Native" strategy. While the React pathway prioritises ecosystem availability and developer velocity, the Slint pathway prioritises runtime efficiency, memory safety, and binary compactness. This strategy entails a complete re-engineering of the Eclipse RCP application, migrating the codebase from Java to Rust and targeting WebAssembly (Wasm) as the deployment runtime.

Slint (formerly SixtyFPS) was selected for this study because its rendering paradigm differs essentially from React's DOM-centric manipulation model. Rather than abstracting HTML elements, Slint renders a proprietary, lightweight widget set directly onto a WebGL canvas. This design choice positions Slint as particularly comparable to the "heavyweight" graphical applications commonly implemented with Eclipse RCP, as it avoids the layout recalculation ("layout thrashing") and styling overhead that are intrinsic to conventional web technologies.

4.4.1 Architectural Mapping: From Widget-Based to Canvas-Based Rendering

The migration from Eclipse SWT to Slint entails a substantial conceptual shift in the specification of the user interface. Eclipse SWT operates as a wrapper around native operating system widgets (e.g., GTK handles on Linux, HWND handles on Windows). In contrast, Slint uses a declarative markup language (.slint) to define the user interface tree, which is subsequently compiled into native Rust code and rendered through a GPU-accelerated pipeline.

Eclipse RCP Concept	Slint/Rust Implementation	Rationale
Composite / Shell	Window / GridLayout	The top-level SWT Shell maps to a Slint <code>Window</code> . Unlike SWT's programmatic layouts, Slint uses declarative constraints similar to CSS Grid but compiled at build time.
SWT Widget (e.g., Text)	Slint Built-in Widgets	Standard widgets (buttons, inputs) are rendered by Slint's internal engine. They look consistent across platforms, resolving the "Operating System variation" issues common in SWT.

Eclipse RCP Concept	Slint/Rust Implementation	Imple-	Rationale
JFace Viewers	for Models		The intricate JFace <code>TableViewer</code> is substituted with declarative <code>for</code> loops in the <code>.slint</code> markup, which iterate over a <code>Model</code> trait provided by a Rust implementation.
Event Loop	Wasm Executor		The explicit SWT event loop is abstracted away. In Wasm, the browser's <code>requestAnimationFrame</code> drives the rendering, while Rust futures handle asynchronous logic.
OSGi Service Registry	Rust Trait Objects		Service discovery is replaced by dependency injection using Rust traits. The <code>dyn Trait</code> pattern provides runtime polymorphism without the overhead of OSGi's dynamic class loading.
Extension Points	Rust Procedural Macros		Using macros for compile-time code generation takes the place of runtime plugin discovery, removing the overhead of dynamically loading extensions while still preserving modularity.
Preferences (IPreferenceStore)	<code>localStorage</code> via <code>web-sys</code>		Preferences are persisted using browser storage APIs accessed through <code>wasm-bindgen</code> , with Rust's type system ensuring type-safe serialization via <code>serde</code> .
Command Framework	Callback Properties		Commands are exposed as callback properties in the Slint UI, connected to Rust handler functions that implement the business logic with compile-time type checking.

Table 4.2: Component Architectural Mapping with Slint

4.4.2 Declarative UI Definition

A significant deviation from the Java-based, imperative approach to UI construction is Slint's domain-specific language (DSL). This structure improves maintainability by localising UI state logic, reducing the tightly coupled imperative logic often found in large SWT composites.

```

1 // .slint file (DSL)
2 import { Button, VerticalBox, LineEdit, HorizontalBox } from
   "std-widgets.slint";
3
4 export component MainForm inherits Window {
5     // 1. Two-way binding property exposed to Rust
6     in-out property <string> user_name: "Developer";
7     in-out property <bool> is_valid: true;
8
9     // 2. Callbacks for business logic
10    callback submit_clicked(string);
11    callback validate_input(string) -> bool;
12
13    VerticalBox {
14        padding: 16px;
15        spacing: 8px;
16
17        Text {
18            text: "Welcome, " + root.user_name;
19            font-size: 18px;
20            font-weight: 700;
21        }
22
23        HorizontalBox {
24            spacing: 8px;
25            LineEdit {
26                text <=> root.user_name; // Bi-directional binding
27                placeholder-text: "Enter your name";
28
29                // 3. Input validation on change
30                edited => {
31                    root.is_valid = root.validate_input(self.text);
32                }
33            }
34            Text {
35                text: root.is_valid ? " " : " ";
36                color: root.is_valid ? #00ff00 : #ff0000;
37            }
38        }
39
40        Button {
41            text: "Submit";
42            enabled: root.is_valid;
43
44            // 4. Signal handling directly in markup
45            clicked => {
46                root.submit_clicked(root.user_name);
47            }
48        }
49    }
50 }

```

Listing 4.5: Slint Declarative Markup

The declarative approach of Slint markup provides several advantages over imperative SWT construction:

- **Compile-Time Validation:** The Slint compiler verifies property types and callback signatures throughout build time, catching errors that would only surface at runtime in Java/SWT.
- **Hot Reloading:** During the development phase, Slint definition files can be modified and reloaded at runtime without requiring recompilation of the entire Rust codebase, thereby substantially reducing the latency of the user interface iteration cycle.
- **Separation of Concerns:** The user interface (UI) structure is explicitly decoupled from the application’s business logic, in contrast to SWT, where layout specifications and event-handling logic are typically co-located within the same Java class.

4.4.3 State Management: Rust Ownership vs. Java GC

The transition from Java’s Garbage Collection (GC) to Rust’s ownership model tackles a common source of technical debt in Eclipse RCP: memory leaks in the UI thread. In the Slint/Rust implementation, state is managed through `Rc<RefCell<T>` patterns designed for single-threaded WebAssembly environments. While this increases implementation complexity caused by the borrow checker, it eliminates entire classes of runtime errors related to null pointers and race conditions.

4.4.3.1 Memory Management Patterns

The Slint/Rust architecture uses several memory management strategies that vary fundamentally from Java’s approach:

```
1 use slint::*;
2 use std::rc::Rc;
3 use std::cell::RefCell;
4
5 // Application state structure
6 #[derive(Clone)]
7 struct AppState {
8     documents: Rc<RefCell<Vec<Document>>>,
9     active_document: Rc<RefCell<Option<usize>>>,
10 }
11
12 impl AppState {
13     fn new() -> Self {
14         Self {
15             documents: Rc::new(RefCell::new(Vec::new())),
16             active_document: Rc::new(RefCell::new(None)),
17         }
18     }
19 }
```

```

20 // Safe mutation through interior mutability
21 fn add_document(&self, doc: Document) {
22     self.documents.borrow_mut().push(doc);
23 }
24
25 // Borrow checker ensures no data races
26 fn get_active_document(&self) -> Option<Document> {
27     let active_idx = *self.active_document.borrow();
28     active_idx.and_then(|idx| {
29         self.documents.borrow().get(idx).cloned()
30     })
31 }
32 }
33
34 // Integration with Slint UI
35 pub fn setup_app() -> MainForm {
36     let app = MainForm::new().unwrap();
37     let state = AppState::new();
38
39     // Clone state for closure (Rc provides cheap cloning)
40     let state_clone = state.clone();
41     app.on_open_document(move |path| {
42         let doc = Document::load(&path);
43         state_clone.add_document(doc);
44     });
45
46     app
47 }

```

Listing 4.6: State Management with Rust Ownership

This pattern provides several guarantees that are impossible in Java:

- **No Null Pointer Exceptions:** The `Option<T>` type forces explicit handling of missing values at compile time.
- **No Data Races:** The borrow checker makes certain that mutable references cannot coexist with other references, preventing concurrent modification bugs.
- **Deterministic Clean-up:** Resources are released as soon as they leave scope, in contrast to Java, where garbage collection introduces non-deterministic pauses.

4.4.4 The Wasm-JS Bridge for External Connectivity

In the Slint pathway, integrating with external services necessitates navigating the WebAssembly sandbox. Since Rust logic exists in a separate linear memory space from the browser’s JavaScript environment, the implementation uses Global Properties and Callbacks as the connection point.

```

1 use wasm_bindgen::prelude::*;
2 use web_sys::window;
3 use serde::{Serialize, Deserialize};
4
5 // Define the data structure for external communication
6 #[derive(Serialize, Deserialize)]
7 struct FormState {
8     user_name: String,
9     is_valid: bool,
10    modified_fields: Vec<String>,
11 }
12
13 #[wasm_bindgen]
14 pub struct WasmBridge {
15     app: slint::Weak<MainForm>,
16 }
17
18 #[wasm_bindgen]
19 impl WasmBridge {
20     #[wasm_bindgen(constructor)]
21     pub fn new(app: &MainForm) -> Self {
22         Self {
23             app: app.as_weak(),
24         }
25     }
26
27     // Expose state to JavaScript
28     #[wasm_bindgen(js_name = getFormState)]
29     pub fn get_form_state(&self) -> JsValue {
30         let app = self.app.upgrade().unwrap();
31         let state = FormState {
32             user_name: app.get_user_name().to_string(),
33             is_valid: app.get_is_valid(),
34             modified_fields: vec![], // Track dirty fields
35         };
36         // Serialize to JS object
37         serde_wasm_bindgen::to_value(&state).unwrap()
38     }
39
40     // Allow external updates
41     #[wasm_bindgen(js_name = setUsername)]
42     pub fn set_user_name(&self, name: String) {
43         let app = self.app.upgrade().unwrap();
44         app.set_user_name(name.into());
45     }
46
47     // Subscribe to changes
48     #[wasm_bindgen(js_name = onStateChange)]
49     pub fn on_state_change(&self, callback: js_sys::Function) {
50         let app = self.app.upgrade().unwrap();
51         let callback_clone = callback.clone();
52         app.on_user_name_changed(move |new_name| {

```

```

53         let this = JsValue::null();
54         let arg = JsValue::from_str(&new_name);
55         let _ = callback_clone.call1(&this, &arg);
56     });
57 }
58 }
59
60 // Initialize the bridge from Rust
61 #[wasm_bindgen(start)]
62 pub fn main() {
63     let app = MainForm::new().unwrap();
64     let bridge = WasmBridge::new(&app);
65
66     // Expose to global scope
67     let window = window().unwrap();
68     let global = js_sys::Reflect::get(&window, &"wasmBridge".into())
69         .unwrap_or(JsValue::undefined());
70     if global.is_undefined() {
71         js_sys::Reflect::set(
72             &window,
73             &"wasmBridge".into(),
74             &JsValue::from(bridge)
75         ).unwrap();
76     }
77
78     app.run().unwrap();
79 }

```

Listing 4.7: Rust to JS Interop via wasm-bindgen

This bridge pattern enables several integration scenarios:

- **External Monitoring:** Development tools or analytics services can query application state without modifying the Rust code.
- **Automated Testing:** Test frameworks are able to drive the UI by calling exposed methods and simulating user operations.
- **Legacy Integration:** Existing JavaScript services can interact with the Wasm module through a clearly defined API surface.

4.4.5 Build Pipeline and Deployment

The Slint/Wasm pathway delivers a significantly different build and deployment model in comparison to traditional web applications:

```
1 # 1. Compile Slint markup to Rust code
2 slint-compiler ui/main.slint --output src/generated/
3
4 # 2. Build Rust to WebAssembly with optimisations
5 cargo build --release --target wasm32-unknown-unknown
6
7 # 3. Optimise Wasm binary (reduces size by 30-50%)
8 wasm-opt -Oz -o output.wasm
9     target/wasm32-unknown-unknown/release/app.wasm
10
11 # 4. Generate JavaScript bindings
12 wasm-bindgen output.wasm --out-dir dist/ --target web
13
14 # 5. Bundle with web assets
15 # Result: ~200KB .wasm + ~50KB .js glue code
```

Listing 4.8: Slint/Wasm Build Pipeline

The deployment artifact is significantly smaller than equivalent React applications:

- **No Node Modules:** The entire application is self-contained in the Wasm binary.
- **Cacheable Binary:** The .wasm binary can be subject to long-lived caching policies, as it is updated exclusively in conjunction with new software releases.
- **CDN-Friendly:** The compact size of the binary aids efficient global distribution via content delivery networks (CDNs), thereby minimising bandwidth utilisation.

4.4.6 Technical Challenges and Solutions

4.4.6.1 Institutional Knowledge Attrition and Documentation Deficiencies

A pervasive challenge in the redesign of software systems that have been in operation for extended periods is the progressive loss of institutional knowledge. The business logic embedded in legacy RCP applications is frequently encapsulated in undocumented “tribal knowledge.” In the absence of the original developers, project teams are often confronted with “black box” components whose intended behaviour can only be inferred through laborious reverse engineering and exhaustive manual testing.

This challenge is especially evident in the Slint/Rust migration pathway due to the following factors:

- **Lack of Incremental Migration:** In contrast to RAP, there is no mechanism to execute legacy Java code alongside Rust within the same runtime environment. Each component must be completely reimplemented prior to any meaningful testing.
- **Type System Mismatch:** Reflection-intensive patterns in Java (common in OSGi-based architectures) are not directly compatible with Rust's static, compile-time type system, entailing considerable architectural refactoring.
- **Non-obvious Dependencies:** Eclipse's dynamic plugin mechanism can conceal dependencies that only materialise at runtime, rendering them difficult to identify through standard integration testing.

To reduce these risks, the implementation adopted the following strategies:

- **Characterisation Testing:** Extensive test suites were developed for the legacy system to capture and formalise its actual behaviour prior to initiating the migration.
- **Parallel Operation:** The legacy and target systems were executed concurrently during the migration phase, with automated evaluation tools used to detect and analyse behavioural divergences.
- **Domain Expert Involvement:** The first cohort of expert users was continuously involved in the development process to validate the correct handling of boundary cases and to provide domain-specific feedback.

4.4.6.2 Feature Parity vs. The Innovation Tax

Redesign projects commonly face a significant "innovation tax", where 70% to 80% of the modernisation budget is spent simply retaining the legacy system's status quo while trying to catch up on the new technology stack. This creates a "feature parity gap" that can last for months or years, during which users must be supported on both the legacy RCP tool and the under-featured web-native version.

In the Slint/Rust context, these problems appear as follows:

- **Widget Library Limitations:** The native widget set provided by Slint is more restricted in scope than that of SWT, necessitating bespoke implementations of specialised user interface components (e.g., syntax-highlighting text editors, tree tables with inline editing capabilities).
- **Platform Integration Constraints:** The deep operating-system integration offered by Eclipse RCP (including system tray interaction, file association handling, and native dialogue support) must be replicated via browser-based APIs, which typically expose a reduced set of functionalities and capabilities.
- **Performance Expectation Mismatch:** End users frequently anticipate that the web-based version will exhibit better performance compared to the

desktop counterpart; however, initial implementations often fail to meet these expectations due to WebAssembly-related overheads and the relative immaturity of optimisation strategies in this environment.

4.4.6.3 Wasm Ecosystem Limitations

Several technical restrictions unique to WebAssembly complicated the implementation:

- **Threading Model:** Wasm threads (via `SharedArrayBuffer`) are not universally supported due to security restrictions, limiting parallelisation opportunities.
- **File System Access:** Direct file input/output operations are not feasible; all persistent storage must be mediated by the browser's application programming interfaces (e.g., the File System Access API), which necessitate explicit user authorisation through permission prompts.
- **Binary Size Sensitivity:** Although a 200 KB payload is relatively modest for a web application, each additional kilobyte can greatly affect performance and data usage, particularly for mobile users. This constraint compels developers to omit potentially valuable libraries in order to adhere to exacting size budgets.
- **Debugging Complexity:** Stack traces produced by WebAssembly (Wasm) are generally less expressive and informative than those available in native execution environments, and existing memory profiling and diagnostic tools for Wasm are less mature compared to the browser's developer tools for JavaScript.

Chapter 5

Integration with IDE Ecosystems

In the legacy Eclipse RCP environment, "packaging" was defined by the OSGi bundle and the feature project, where `plugin.xml` files declared the tool's contributions to the workbench. Modern web IDEs have replaced this with more flexible models that vary by platform. The migration strategy must take into account two distinct integration approaches: the VS Code Extension API (sandboxed, runtime-installable) and Theia Extensions (compile-time dependency injection with deep IDE access).

The decision to establish a unified hosting strategy is driven by the requirement to concurrently support multiple frontend technologies during the migration phase. A typical Eclipse RCP tool suite may comprise highly interactive diagram editors, form-based property views, and data-intensive tabular components. Migrating all these parts using a single framework is not necessarily optimal; for instance, a diagram editor may derive particular advantages from the React component model, whereas a performance-critical utility panel may be more appropriately implemented with Slint compiled to WebAssembly (Wasm).

This chapter analyses two complementary integration strategies:

- **WebView-based integration** (Section 5.1): A cross-IDE approach compatible with both VS Code and Theia, relying on sandboxed iframes and structured message-passing protocols for interaction between the extension and the host environment.
- **Integration via native Theia extensions** (Section 5.2): A Theia-specific mechanism that exploits the Inversify.js dependency injection framework to enable first-class, deeply integrated interaction with the IDE workbench.

A strong migration strategy primarily adopts the VS Code Extension model to maximise cross-platform portability, while reserving native Theia extensions for use cases which require deep workbench integration or advanced white-labelling and product-branding capabilities.

5.1 Integration based on WebView

The technical implementation of the cross-platform hosting strategy is based on a generic WebView Bridge. In modern web-based IDEs, a "WebView" is an isolated context—essentially a sandboxed iframe—that allows an extension to render arbitrary HTML content. However, the raw API provided by the IDE is a basic building block that calls for extensive orchestration to support professional-grade tools.

The bridge acts as an architectural wrapper that abstracts the IDE-specific details of WebView management. Using the Template Method Pattern, the implementation encapsulates the complications of resource loading, URI resolution, security policy enforcement, and lifecycle synchronisation in an abstract base class (`AbstractToolPanel`). This allows specific frontend pathways to focus purely on domain logic while inheriting a standardised integration with the workbench.

5.1.1 URI Resolution and Local Resource Access

One of the main issues addressed by the generic wrapper concerns the restriction on loading local resources. For security reasons, WebViews are unable to access files from the user's filesystem via the standard `file://` protocol. To overcome this limitation, the bridge intercepts requests for local assets—such as bundled JavaScript, CSS, or image files—and rewrites them into IDE-specific URIs by invoking the `webview.asWebviewUri()` method. This URI resolution mechanism is essential for uniting modern build systems, including Vite and Webpack, as it guarantees that relative paths to internal assets remain valid within the constrained, sandboxed execution environment.

5.1.2 Dynamic HTML Content Generation and Security

The generic wrapper uses a template-based approach to dynamically generate the WebView's HTML content. This process applies strict security standards compliant with the IDE's Content Security Policy (CSP):

- **Nonce Generation:** To prevent Cross-Site Scripting (XSS), the bridge generates a unique, cryptographic nonce for each session. This nonce is injected into the HTML `<meta>` tag and all script elements, making sure that only authorised scripts loaded by the extension can be executed.
- **Theme Synchronisation:** The bridge provides CSS variables aligned with the IDE's active theme colours (e.g., `-vscode-editor-foreground`), making sure the migrated tool visually corresponds to the surrounding environment.
- **Environment Detection:** The bridge inspects the `ExtensionMode` to decide whether to load the production build from the `dist` directory or connect to a local development server, therefore supporting Hot Module Replacement (HMR).

5.1.3 The Communication Protocol: Type-Safe JSON-RPC

The physical separation of the frontend (WebView) and backend (Extension Host) creates a "version barrier" similar to that found inside distributed systems. Unlike Eclipse RCP, which allowed synchronous method calls between the UI and core services, the web-based model relies on asynchronous message passing.

To reduce the complexity of raw message handling, the implementation utilises a formal Remote Procedure Call (RPC) protocol. This effectively implements a simplified Actor Model, where the UI and Logic run in isolated environments and communicate via immutable messages. The RPC layer provides a type-safe abstraction, defining shared TypeScript interfaces for the API contract to ensure compile-time validation of all frontend-backend interactions.

```

1 // Shared Contract (imported by both React and Extension Host)
2 export interface RpcMethods {
3   saveFile(path: string, content: string): Promise<void>;
4   getWorkspaceFiles(): Promise<string[]>;
5 }
6
7 // Frontend RPC Client (Running in React/WebView)
8 class RpcClient {
9   private pendingRequests = new Map<string, { resolve: Function,
10     reject: Function }>();
11   private vscode = acquireVsCodeApi(); // VS Code global API
12
13   constructor() {
14     // Listener acting as the "Mailbox" for the Actor Model
15     window.addEventListener('message', (event) => {
16       const message = event.data;
17       if (message.type === 'RPC_RESPONSE') {
18         const { requestId, result, error } = message;
19         const promise = this.pendingRequests.get(requestId);
20         if (promise) {
21           if (error) promise.reject(new Error(error));
22           else promise.resolve(result);
23           this.pendingRequests.delete(requestId);
24         }
25       }
26     });
27   }
28
29   /**
30    * Generic wrapper to call backend methods with Type Safety
31    */
32   public call<K extends keyof RpcMethods>(
33     method: K,
34     ...args: Parameters<RpcMethods[K]>
35   ): ReturnType<RpcMethods[K]> {
36     const requestId = crypto.randomUUID();
37
38     // Send message to Extension Host

```

```

38     this.vscode.postMessage({ type: 'RPC_REQUEST', requestId, method,
39                               args });
40
41     // Return a Promise that resolves when the response arrives
42     return new Promise((resolve, reject) => {
43         this.pendingRequests.set(requestId, { resolve, reject });
44     }) as ReturnType<RpcMethods[K]>;
45 }

```

Listing 5.1: Type-Safe RPC Bridge Implementation

5.1.3.1 Mechanism of the RPC Lifecycle

The communication protocol follows a rigorous request-response lifecycle to handle the asynchronous nature of the bridge:

1. **Request Construction:** The bridge generates a unique Request ID (UUID) and packages the method name and arguments.
2. **Promise Management:** A Promise is instantiated, and its resolve/reject callbacks are registered in a `pendingRequests` map.
3. **Transmission:** The payload is dispatched to the Extension Host via `vscode.postMessage()`.
4. **Execution & Response:** The Extension Host receives the payload, forwards it to the corresponding service (for example, a file system provider), awaits the outcome, and then returns a response message that includes the stated Request ID.
5. **Resolution:** The frontend bridge looks up the previously stored callback by the Request ID, fulfils the Promise, and passes the resulting data back to the initiating component.

5.1.4 Workspace and Filesystem Interoperability

Providing WebView access to the IDE workspace is a fundamental requirement for any migrated Eclipse RCP tool. Since WebViews are restricted from direct disk access due to the browser sandbox, all filesystem operations must be proxied through the Extension Host using the `vscode.workspace.fs` API.

The implementation of workspace interoperability via remote procedure calls (RPC) enables several improved capabilities:

- **ACID-Compliant File Operations:** The RPC protocol assures that file persistence operations exhibit transactional behaviour consistent with ACID properties. The frontend transmits the fully updated model state, while the Extension Host executes the corresponding atomic write operation. In the presence of workspace restrictions, the Extension Host returns clear error

feedback, therefore preserving consistency and preventing partial or invalid state changes.

- **Virtual Workspace Support:** Modern IDEs often work with virtual filesystems (e.g., GitHub Repositories or SSH protocol remotes). The RPC-based integration guarantees that the tool is "filesystem-agnostic," as the Extension Host transparently handles the translation of virtual URIs.
- **Workspace Metadata Pushing:** Upon initialisation, the Extension Host pushes workspace folder structures and configuration settings to the WebView, allowing the tool to immediately adapt its UI context (e.g., specific file icons or project filters).

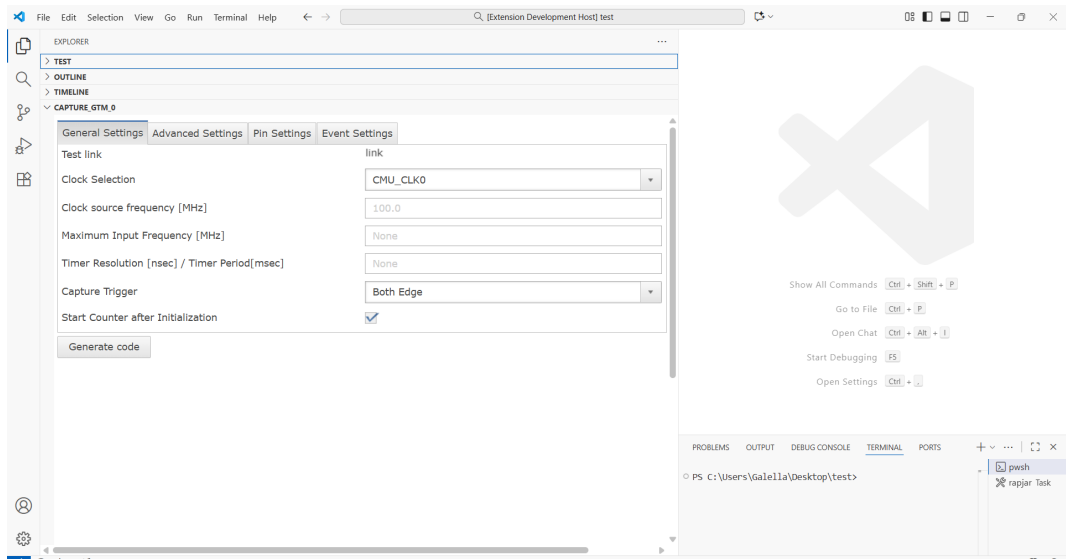


Figure 5.1: Tool application running in a VS Code WebView

5.1.5 The WebAssembly Bootstrapping Process

The embedding of a Slint-based tool into an IDE such as VS Code or Theia differs essentially from the React approach. Because the application logic is compiled into a WebAssembly (Wasm) binary, the "WebView Bridge" must act as a specialised Wasm Loader and FFI (Foreign Function Interface) host.

In contrast to a JavaScript bundle, which is executed immediately upon parsing by the browser, a Slint (Wasm) binary must be retrieved, compiled, and instantiated in an asynchronous manner. The newly introduced integration layer is responsible for initialising the Wasm linear memory and for injecting the JavaScript "glue" code required to enable Rust code to interoperate with the DOM.

- **Binary Loading:** The VS Code extension retrieves the `.wasm` binary from its installation folder and translates it into a secure URI via the `asWebviewUri` mechanism.

- **Memory Allocation:** At startup, the host reserves a linear memory buffer. This buffer serves as the sole memory region accessible to the Slint engine, enforcing a sandboxed execution environment comparable to the Eclipse RCP "Security Manager" but with markedly decreased overhead.

5.1.5.1 The Communication Protocol: Async RPC over FFI

Because the Rust logic exists in a separate memory space from the IDE's JavaScript environment, direct function calls between the UI and the VS Code API are impossible. Integration uses a Foreign Function Interface (FFI) to bridge the gap.

The following TypeScript implementation presents the host-side component of this bridge, which is responsible for instantiating the Slint module and propagating requests to the VS Code API.

```

1  import init, { main_func } from './wasm/slnt_app.js';
2  const vscode = acquireVsCodeApi();
3  /**
4   The FFI Provider: These functions are "injected" into the
5   WebAssembly instance so that Rust can call them.
6   */
7  const ffiProvider = {
8   // Rust calls this to write a file to the workspace
9   save_file_to_disk: (path: string, content: string) => {
10     vscode.postMessage({ command: 'saveFile', payload: { path,
11       content } });
12   },
13   // Rust calls this to log telemetry to the VS Code Output channel
14   log_info: (message: string) => {
15     vscode.postMessage({ command: 'log', message });
16   }
17 };
18
19 async function startTool() {
20   // 1. Fetch and compile the Wasm binary
21   const wasmSource = await init();
22   // 2. Expose the FFI to the global window for wasm-bindgen to find
23   (window as any).rpc_bridge = ffiProvider;
24   // 3. Start the Slint Event Loop
25   main_func();
26 }
27
28 // Listen messages FROM the VS Code Extension Host (e.g. File Data)
29 window.addEventListener('message', event => {
30   const { command, data } = event.data;
31   if (command === 'fileLoaded') {
32     // Call back into the Wasm module to update the Slint state
33     (window as any).update_slint_model(data);
34   });
35
36 startTool();

```

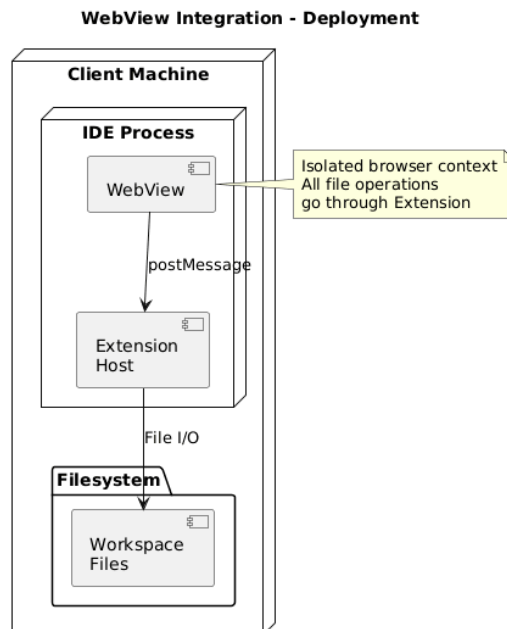
Listing 5.2: Wasm-to-VS Code Integration Bridge**5.1.5.2 The Virtual Filesystem (VFS) Layer**

To bridge the gap between Eclipse’s `IResource` API and the browser sandbox, the implementation delivers a VFS Layer written in Rust. This layer functions as a proxy; when the Slint UI requests a resource, the VFS checks a local Wasm-side cache before triggering an asynchronous RPC call through the bridge shown in Listing 5.2. This makes certain that the UI stays responsive (60 FPS) even when waiting for slow disk I/O from the host IDE.

5.1.5.3 Technical Challenge: Serialisation and the "Wasm Boundary"

A major difference in this integration pathway is the Serialisation Tax. In the React pathway, objects are passed as native JavaScript references. In the Slint/Wasm pathway, every message must be serialised to a string or byte array to cross the Wasm boundary.

For high-frequency update scenarios—such as maintaining synchronisation between an Eclipse-style “Outline View” and a text editor—the associated overhead was reduced by using ‘`SharedArrayBuffer`’ objects. This mechanism enabled the JavaScript host environment and the Rust binary to access a common contiguous memory region directly, therefore supporting the processing of thousands of updates per second while eschewing the latency introduced by JSON serialisation.

**Figure 5.2:** Deployment diagram - WebView

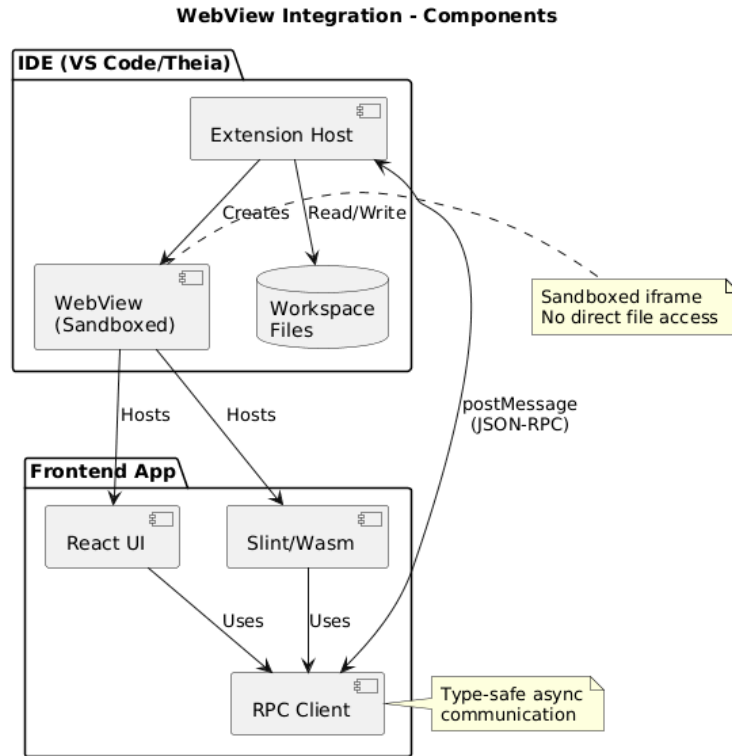


Figure 5.3: Component diagram - WebView

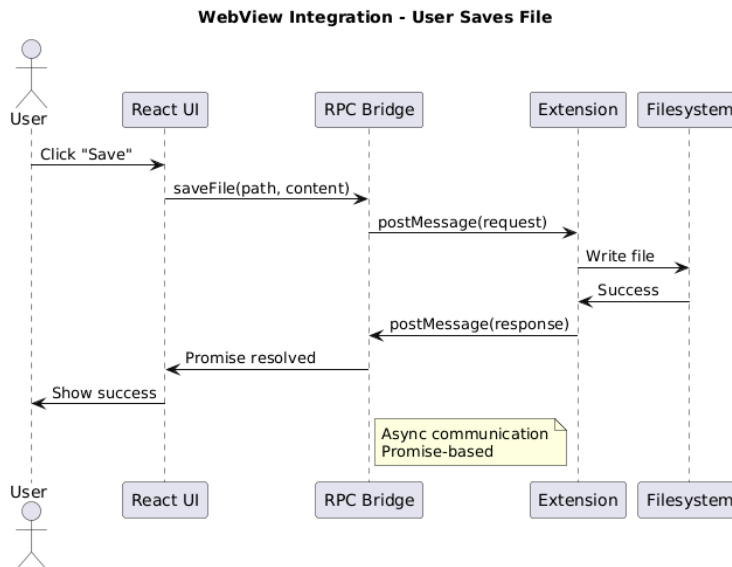


Figure 5.4: Sequence diagram - WebView

5.2 Integration of native Theia extensions

Although the WebView-based approach delivers maximum portability between VS Code and Theia, it introduces architectural overhead due to message serialisation and sandbox isolation. For Theia-specific deployments, particularly in white-labelled or domain-targeted IDE scenarios, native Theia Extensions offer a fundamentally different integration model that more closely resembles the Eclipse RCP architecture.

5.2.1 Architectural Foundation: Inversify.js Dependency Injection

Theia's software architecture is based on Inversify.js, a TypeScript-based implementation of dependency injection that closely resembles the Eclipse OSGi service registry. In contrast to Visual Studio Code extensions, which are executed in isolated Node.js processes and interact with the host application via inter-process communication (IPC), Theia extensions are statically compiled into the integrated development environment (IDE) and operate within the same dependency injection container as the core workbench.

This architectural design choice provides several advantages for the migration of Eclipse Rich Client Platform (RCP) tools:

- **Direct API Access:** Extensions can inject and extend core services (e.g., `WorkspaceService`, `FileService`, `EditorManager`) without incurring the overhead involved with remote procedure calls (RPC).
- **Synchronous Invocation:** Interactions between extensions and core services are realised as direct TypeScript function calls, thereby eliminating (de-)serialisation costs and associated latency.
- **Compile-Time Type Safety:** The TypeScript compiler enforces service interface contracts at build time, allowing the detection of integration errors prior to runtime deployment.
- **Lifecycle Integration:** Extensions can participate in Theia's lifecycle management (`FrontendApplicationContribution`, `BackendApplicationContribution`), providing fine-grained control over initialisation order and startup behaviour.

5.2.2 Widget-Based UI Integration

Rather than rendering content in an isolated `WebView`, native Theia extensions create first-class widgets that integrate fluently with the workbench layout. This approach replicates Eclipse RCP's `ViewPart` and `EditorPart` abstractions.

5.2.2.1 Example: Migrating an Eclipse RCP View to a Theia Widget

The following example shows how to migrate a typical Eclipse RCP property view to a native Theia widget:

```
1 import { injectable, inject } from '@theia/core/shared/inversify';
2 import { ReactWidget } from
   '@theia/core/lib/browser/widgets/react-widget';
3 import { WorkspaceService } from '@theia/workspace/lib/browser';
4 import { FileService } from
   '@theia/filesystem/lib/browser/file-service';
5 import * as React from 'react';
6
7 @injectable()
```

```

8 export class PropertyViewWidget extends ReactWidget {
9   static readonly ID = 'property-view:widget';
10  static readonly LABEL = 'Properties';
11
12  @inject(WorkspaceService)
13  protected readonly workspaceService!: WorkspaceService;
14
15  @inject(FileService)
16  protected readonly fileService!: FileService;
17
18  private modelData: any = null;
19
20  constructor() {
21    super();
22    this.id = PropertyViewWidget.ID;
23    this.title.label = PropertyViewWidget.LABEL;
24    this.title.caption = PropertyViewWidget.LABEL;
25    this.title.closable = true;
26    this.title.iconClass = 'fa fa-list-alt';
27
28    // Subscribe to workspace changes (analogous to Eclipse
29    // IResourceChangeListener)
30    this.workspaceService.onWorkspaceChanged(() =>
31      this.refreshModel());
32  }
33
34  protected async refreshModel(): Promise<void> {
35    const roots = await this.workspaceService.roots;
36    if (roots.length > 0) {
37      const modelFile = roots[0].resource.resolve('model.json');
38      const content = await this.fileService.read(modelFile);
39      this.modelData = JSON.parse(content.value);
40      this.update(); // Trigger React re-render
41    }
42  }
43
44  protected render(): React.ReactNode {
45    return (
46      <div className='property-view-container'>
47        <h3>Model Properties</h3>
48        {this.modelData ? (
49          <PropertyGrid data={this.modelData}
50            onSave={(updated) => this.saveModel(updated)} />
51        ) : (
52          <div>No model loaded</div>
53        )}
54      </div>
55    );
56  }
57
58  protected async saveModel(updatedData: any): Promise<void> {
59    const roots = await this.workspaceService.roots;
60    const modelFile = roots[0].resource.resolve('model.json');

```

```

59     await this.fileService.write(modelFile,
JSON.stringify(updatedData, null, 1));
60 }
61 }

```

Listing 5.3: Native Theia Widget Implementation

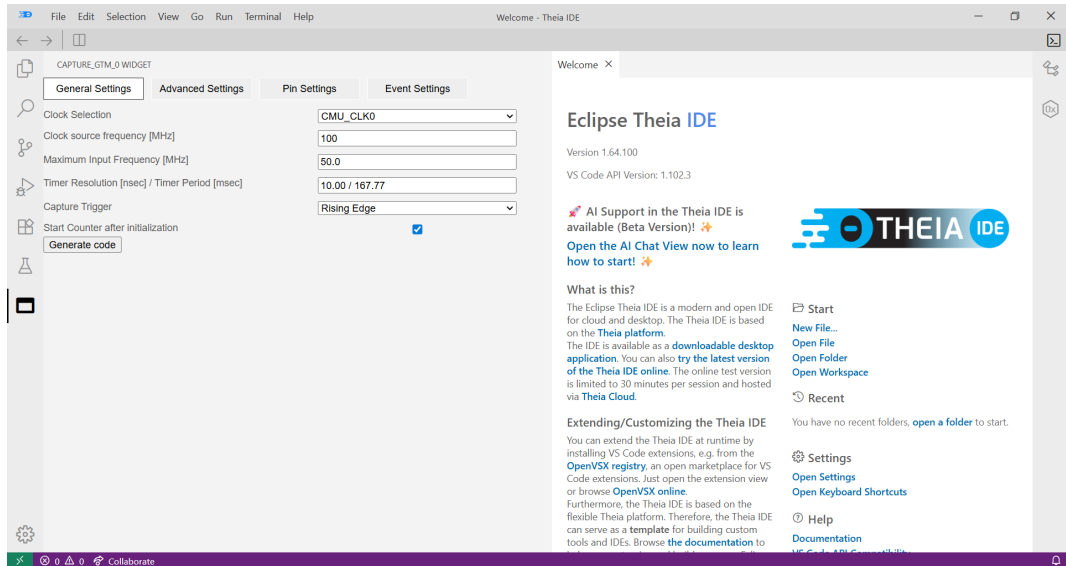


Figure 5.5: Tool application running in a Theia extension

5.2.2.2 Key Architectural Differences from the WebView Approach

Aspect	WebView Integration	Native Theia Widget
Service Access	Async RPC via postMessage	Direct injection via @inject
Type Safety	Runtime validation only	Compile-time TypeScript checking
Performance	Serialisation overhead	Zero-copy function calls
Portability	Works in VS Code + Theia	Theia-only
Debugging	Separate DevTools context	Unified debugging experience
Theming	Manual CSS variable injection	Automatic theme propagation

Table 5.1: Comparison of Integration Approaches

5.2.3 Dependency Injection Module Configuration

To register the widget with Theia’s DI container, the extension must provide a frontend module that binds the widget class and contributes it to the workbench:

```

1 import { ContainerModule } from '@theia/core/shared/inversify';

```

```

2 import { PropertyViewWidget } from './property-view-widget';
3 import { WidgetFactory, bindViewContribution } from
  '@theia/core/lib/browser';
4 import { PropertyViewContribution } from
  './property-view-contribution';
5
6 export default new ContainerModule(bind => {
7   // Bind the widget as a singleton
8   bind(PropertyViewWidget).toSelf().inSingletonScope();
9
10  // Register widget factory
11  bind(WidgetFactory).toDynamicValue(ctx => ({
12    id: PropertyViewWidget.ID,
13    createWidget: () =>
14      ctx.container.get<PropertyViewWidget>(PropertyViewWidget)
15  })).inSingletonScope();
16
17  // Bind the contribution that adds menu/command entries
18  bindViewContribution(bind, PropertyViewContribution);
19 });

```

Listing 5.4: Theia Frontend Module

The `PropertyViewContribution` class (not shown) would implement `FrontendApplicationContribution` to register commands and menu items, analogous to Eclipse RCP's `plugin.xml` contributions.

5.2.4 Integration of Backend Services

For tools that need server-side capabilities (such as code generation or model validation), native Theia extensions can declare backend services that execute within the Node.js process. The frontend communicates with these services using JSON-RPC, and in contrast to the `WebView` method, the protocol is derived automatically from the TypeScript interfaces.

```

1 // 1. Shared interface (common package)
2 export const ModelValidatorPath = '/services/model-validator';
3 export const ModelValidator = Symbol('ModelValidator');
4
5 export interface ModelValidator {
6   validate(modelUri: string): Promise<ValidationResult>;
7 }
8
9 // 2. Backend implementation (backend package)
10 @injectable()
11 export class ModelValidatorImpl implements ModelValidator {
12   @inject(FileService)
13   protected readonly fileService!: FileService;
14
15   async validate(modelUri: string): Promise<ValidationResult> {
16     const content = await this.fileService.read(new URI(modelUri));
17     // Perform validation logic

```

```

18     return { errors: [], warnings: [] };
19   }
20 }
21
22 // 3. Frontend proxy (automatically generated)
23 // The frontend simply injects ModelValidator and calls methods as if
    local
24 @injectable()
25 export class PropertyViewWidget extends ReactWidget {
26   @inject(ModelValidator)
27   protected readonly validator!: ModelValidator;
28
29   protected async validateCurrentModel(): Promise<void> {
30     const result = await
31     this.validator.validate(this.currentModelUri);
32     this.displayValidationResults(result);
33   }
34 }

```

Listing 5.5: Backend Service with Auto-Generated RPC

The Theia framework automatically handles:

- Proxy generation for the frontend client.
- Implementation of the JSON-RPC protocol.
- Management of the connection lifecycle.
- Error transmission and retry logic.

5.2.5 Filesystem Integration: Direct API Access

Unlike the WebView approach, which requires proxying all filesystem operations through the Extension Host, native Theia widgets have direct access to the `FileService` API:

```

1 @injectable()
2 export class ModelEditorWidget extends ReactWidget {
3   @inject(FileService)
4   protected readonly fileService!: FileService;
5
6   @inject(FileSystemWatcher)
7   protected readonly watcher!: FileSystemWatcher;
8
9   async init(): Promise<void> {
10     // Watch for changes to model files (analogous to Eclipse
        IResourceChangeListener)
11     this.watcher.onFilesChanged(event => {
12       const modelChanges = event.changes.filter(c =>
13       c.resource.path.ext === '.model'
14     });
15     if (modelChanges.length > 0) {
16       this.refreshEditor();
17     }
18   }
19 }

```

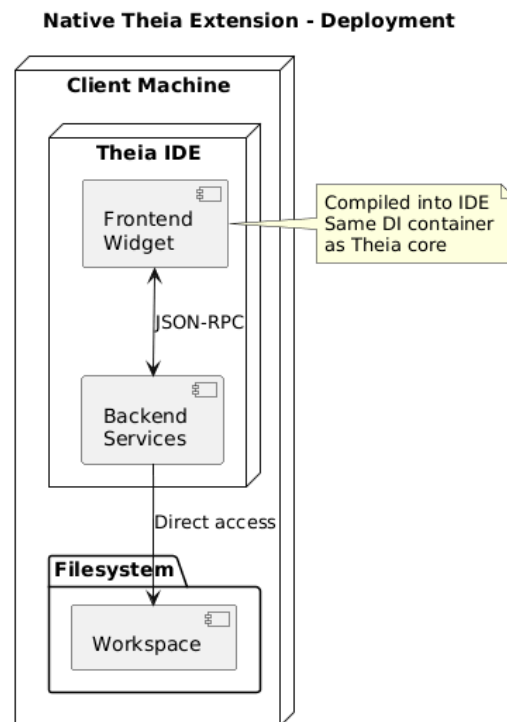
```

17     }
18   });
19 }
20
21 protected async saveModel(content: string): Promise<void> {
22   const uri = new URI(this.currentModelPath);
23
24   // ACID-compliant write with automatic conflict resolution
25   try {
26     await this.fileService.write(uri, content);
27   } catch (error) {
28     if (FileOperationError.isFileOperationError(error) &&
29       error.fileOperationResult ===
30       FileOperationResult.FILE_MODIFIED_SINCE) {
31       // Handle conflict (analogous to Eclipse's IFile.setContents
32       // with FORCE flag)
33       await this.resolveConflict(uri, content);
34     }
35   }
36 }

```

Listing 5.6: Direct Filesystem Operations in Theia

This direct access eliminates RPC overhead and provides synchronous error handling, making the code more maintainable and closer to the authentic Eclipse RCP implementation.

**Figure 5.6:** Deployment diagram - Theia extension

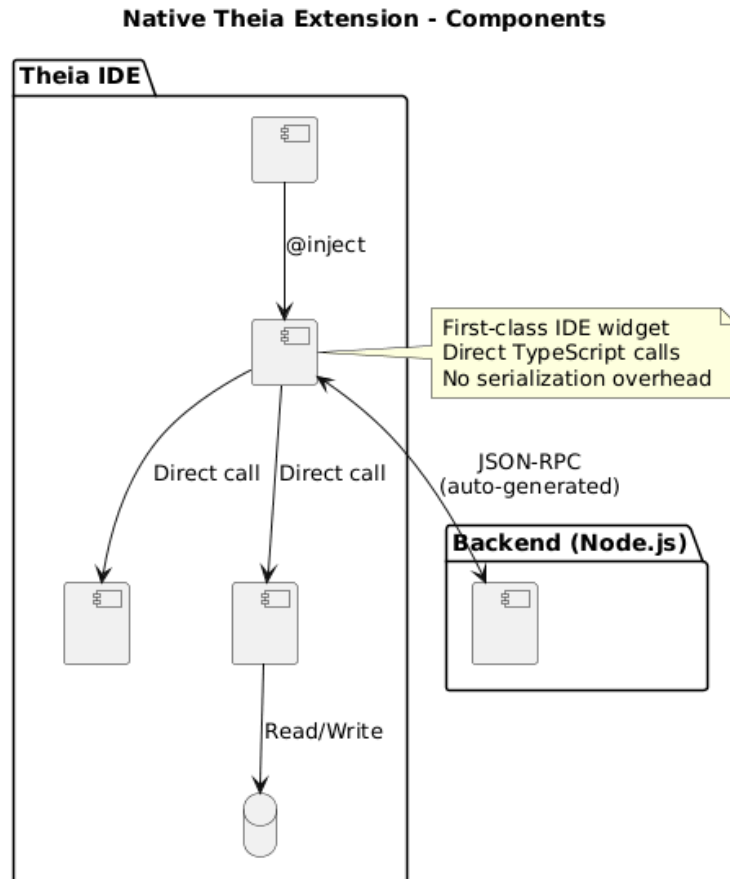


Figure 5.7: Component diagram - Theia extension

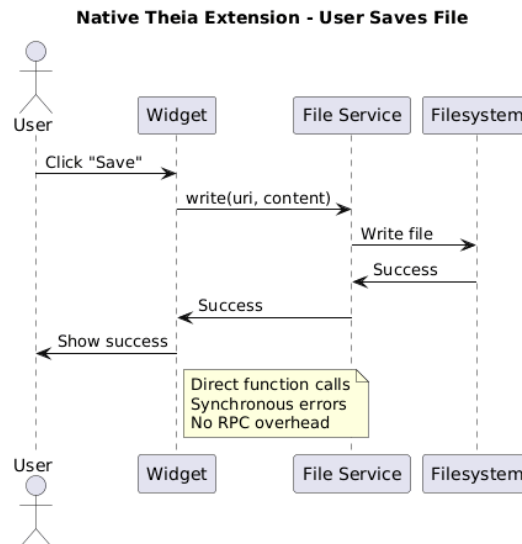


Figure 5.8: Sequence diagram - Theia extension

5.3 The Exception: Eclipse RAP

Within the field of Eclipse RCP migration options, the Remote Application Platform (RAP) is something of an outlier. Unlike the React and Slint approaches, which are

client-native and plug into the IDE's webview using standard web technologies, RAP is based on a fundamentally different architectural model, which introduces major challenges for integration.

5.3.1 Fundamental Architectural Divergence

RAP was designed to allow developers to build web applications using the same Java programming paradigms and APIs (SWT/JFace) found in Eclipse RCP. However, RAP implements a "Thin Client" architecture in which the application logic and the entire UI state reside on a central Java server. The client-side is a thin JavaScript engine that merely renders widgets and forwards user events to the server via Ajax requests.

This difference leads to a basic mismatch when working with modern IDE extensions. Whereas a VS Code or Theia extension executes inside a Node.js process, a RAP application depends on an active Java Virtual Machine (JVM) and a JEE servlet container (for example, Jetty or Tomcat). As a result, a RAP-based solution cannot simply be shipped as a static collection of files within the extension; it must rely on a continuously running server-side environment.

5.3.2 Complexity of RAP-IDE Integration

The embedding of RAP into a VS Code or Theia panel is significantly more complex than the client-side frameworks for several reasons:

- **Lifecycle Management:** The extension must manage the lifecycle of a Java process. This often requires the extension to include an embedded Java runtime and a headless RAP server (Sidecar Pattern).
- **Network Proxying:** Because the RAP server operates as an independent process (typically bound to a local port), the WebView needs to direct its source URL to this local endpoint. This can lead to possible complications such as port collisions and restrictions imposed by network security policies.
- **State Synchronisation:** Unlike the RPC protocol used by React to sync with the Extension Host, RAP's state is managed by the server-side RWT (RAP Widget Toolkit). Synchronising this state with the IDE's workspace requires an additional bridge between the Java RAP server and the Node.js Extension Host.
- **Latency and Performance:** RAP is highly sensitive to network latency because every exchange involves a round-trip to the Java server. In a remote IDE environment (like Codespaces or Theia Cloud), this can lead to a degraded user experience compared to the instantaneous reactivity of a client-side React app.

5.3.3 Integration Strategy: Headless Java Sidecar

The main functionalities needed from a VS Code extension in order to run a Spring RAP application are the following:

- **Launch a VS Code task** that allows the JAR file that contains the Spring RAP application to be run. This background task starts the web application on the localhost.
- **Host an iframe** that points to the Spring RAP URL.
- **Establish a message bridge** using VS Code APIs between the Spring RAP web client and the VSCode host extension to synchronise file state between the local VSCode editor and the remote RAP views.

To start the `syncExec()` call, the user interface thread must be "awakened". This can be achieved by re-rendering the RAP UI, for example, when the user interacts with the UI thread to change tabs; alternatively, by forcefully calling the `readAndDispatch()` method from the UI thread.

As the documentation reports, `readAndDispatch()`: "reads an event from the event queue, dispatches it correctly, and returns true if there is potentially more work to do, or false if the caller can sleep until another event is placed on the event queue."

In addition to checking the system event queue, this method also checks whether any inter-thread messages (created by `syncExec()` or `asyncExec()`) are waiting to be processed and, if so, handles them before returning.

This synchronisation problem can be solved by adding a button to the RAP UI that updates the RAP UI and calls `readAndDispatch()` on the UI thread.

```

1 Button b = new Button(parent, SWT.PUSH);
2 b.setText("Generate code");
3 b.addSelectionListener(new SelectionAdapter() {
4     @Override
5     public void widgetSelected(SelectionEvent e) {
6         while(!RAPEntryPoint.done)
7             while(RAPEntryPoint.getParent().readAndDispatch());
8         RAPEntryPoint.done = false;
9     }
10 });

```

At this stage, the Spring RAP can respond to an API request. But, on the other hand, how can the VS Code extension send an API request to the Spring RAP application?

One potential approach is to repeatedly poll the API via a fetch request and keep doing so until the end user ultimately decides to submit it. This approach is highly inefficient. Therefore, a **WebSocket** connection can be set up between the Spring RAP application and the VS Code extension.

On the Spring RAP side, a message is sent to the external tool endpoint whenever the end user initiates a fetch request. This can occur, for instance, by clicking the previously created button. In this way, the button not only updates the UI thread but also triggers a fetch API request.

On the VS Code extension side, an asynchronous process keeps listening for new messages. When a new message arrives, an API GET request is sent to the mapped Spring RAP URL (e.g. `https://localhost:<PORT>/rap/api`).

After the button is pressed and the Spring RAP sends back the values of the widgets via the REST API, the VS Code extension handles the creation/modification of a new file inside the user's workspace. This is made possible by VS Code's built-in commands, which can be run via `vscode.commands.executeCommand` API.

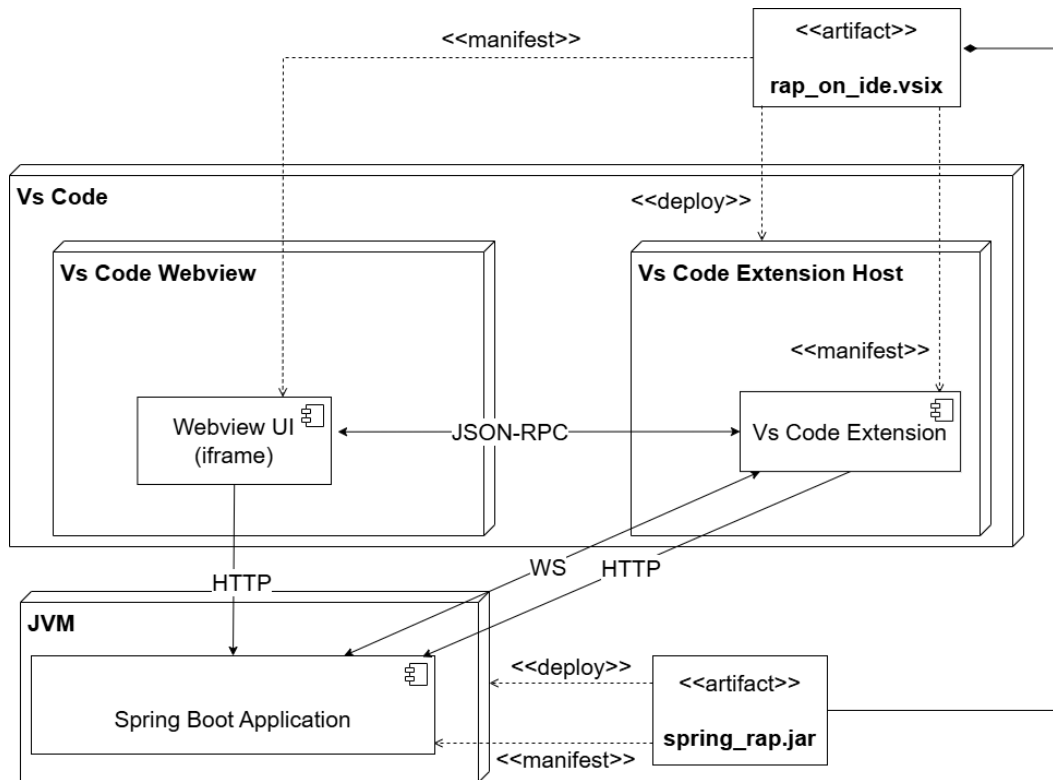


Figure 5.9: Deployment diagram - Eclipse RAP and VS Code

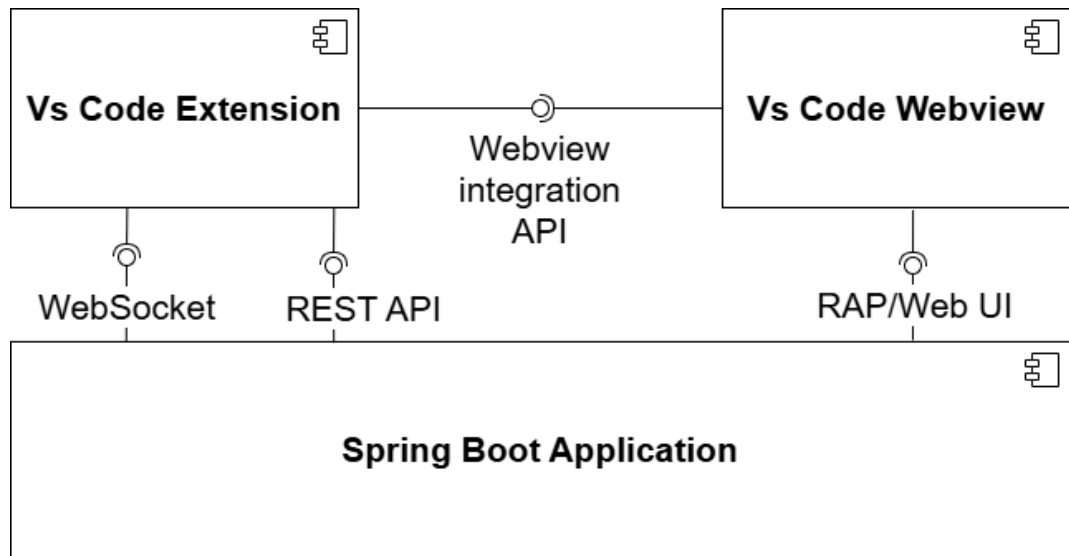


Figure 5.10: Component diagram - Eclipse RAP and VS Code

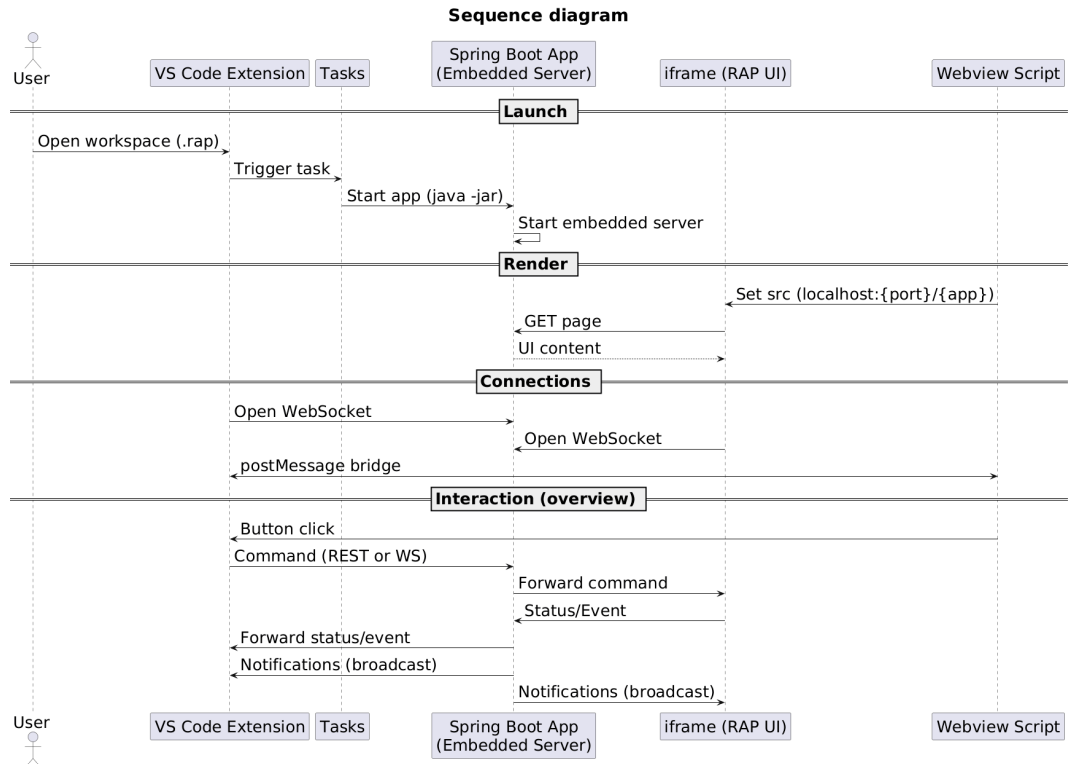


Figure 5.11: Sequence diagram - Eclipse RAP and VS Code

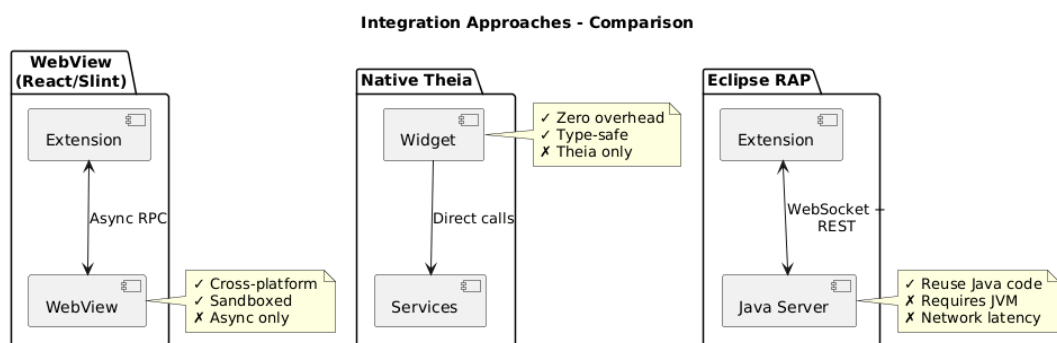


Figure 5.12: Comparison architecture diagrams

Chapter 6

Empirical Analysis and Comparative Evaluation

The transformation of legacy Eclipse Rich Client Platform (RCP) applications into modern web-native tools amounts to a fundamental re-architecture of the software delivery model. This chapter presents empirical findings from the migration of two representative automotive development tools: a parameter-dense clock configuration form and a graphically advanced Automotive Bus Editor. By conducting a rigorous empirical investigation of three migration pathways—Eclipse Remote Application Platform (RAP), React, and WebAssembly with Slint—this study validates a perceptive decision-making framework which balances code reuse objectives with the technical efficiencies of native web architectures.

The analysis is organised into three evaluation phases:

1. A universal maintainability review that determines production readiness across every pathway.
2. An architectural analysis that identifies cloud-native compatibility limitations and directs strategic exclusions.
3. Focused comparison of web-native pathways amid differentiated use case conditions.

6.1 Subject Applications: Architectural Profile and Modernisation Goals

The effectiveness of a modernisation strategy depends heavily on the architectural characteristics and functional requirements of the target application. To ensure the generalisability of the results, this study selects two applications that represent primary interface and logic patterns found in professional engineering environments.

6.1.1 Case Study 1: Clock Configuration Application (Standard Form-Based Tool)

The first subject is a high-density application developed by Infineon Technologies for configuring system and peripheral clocks in automotive microcontrollers. In its traditional Eclipse RCP form, the application manages hundreds of individual calibration parameters organised inside hierarchical categories. The complexity does not lie in graphical fidelity, but in the depth of the validation logic that secures compliance with ISO 26262 standards for functional safety.

Technical Characteristics:

- Complex hierarchy of Standard Widget Toolkit (SWT) layouts employing deeply nested composite widgets.
- Tight coupling with the Eclipse Modelling Framework (EMF) for the representation, storage, and management of parameter metadata.
- Safety-critical Java-based validation logic implementing and enforcing hardware-specific constraints.
- Hierarchical, multi-stage dependency resolution mechanism for the computation of clock frequencies.

Modernisation Objectives:

- Preserve the semantic correctness and operational soundness of the existing Java validation logic, with an emphasis on maximal code reuse.
- Provide a responsive, web-based interface for parameter specification and editing.
- Remove the need for local desktop installations in order to support geographically distributed engineering teams.
- Ensure functional equivalence with the legacy tooling, eschewing any regression in available features.

Representative Use Case: This application showcases most of the Eclipse RCP enterprise tooling spectrum: a moderately complex user interface, substantial business logic, conventional CRUD operations, and well-defined validation rules. Its performance requirements match typical business systems (user engagements completing in under 300 ms), making it a natural reference point for evaluating broad migration approaches.

6.1.2 Case Study 2: Automotive Bus Monitor (Graphics-Intensive Tool)

The second subject is an automotive communication bus evaluation tool designed for live monitoring and simulation of CAN, LIN, and Automotive Ethernet traffic.

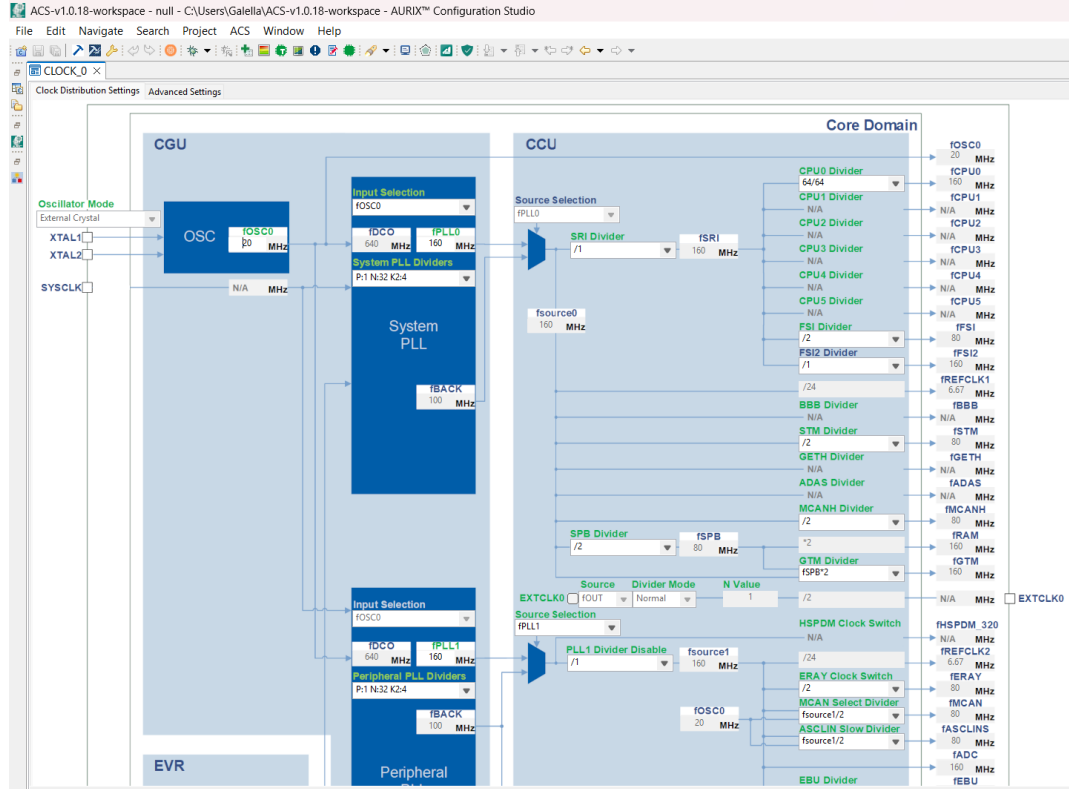


Figure 6.1: Clock Configuration Application - Legacy Eclipse RCP Implementation

Created specifically for this thesis, it falls into the "high-performance" class of automotive instruments that need advanced graphical user interfaces.

Technical Characteristics:

- Real-time waveform rendering for live signal visualisation (targeting 60 FPS)
- Intricate signal packet decoding with millisecond-level latency requirements
- Interactive 3D visualisations of cockpit human-machine interfaces (HMI)
- Legacy implementation using the Graphical Editing Framework (GEF) and GMF (Graphical Modelling Framework)
- Canvas-based editor with custom render pipelines

Modernisation Objectives:

- Attain desktop-comparable rendering performance, specifically a sustained frame rate of 60 frames per second (FPS), for signal visualisation within a web browser environment.
- Minimise end-to-end latency in user interface-backend communication under conditions of high-frequency data updates.
- Migrate to a cloud-compatible system architecture that supports synchronous and asynchronous collaborative analysis sessions.

- Preserve high-fidelity rendering quality to ensure reliable evaluation of safety-critical signals.

Representative Use Case: The application serves as a validation platform for assessing migration strategies of computationally heavy tools in which rendering performance constitutes a primary constraint. In contrast to the clock configuration tool, the success metrics in this context prioritise frame rate stability, memory utilisation attributes under elevated load, and CPU consumption associated with graphical processing tasks. This specialised scenario is designed to isolate and characterise performance differentials between implementations based on React and those using WebAssembly.

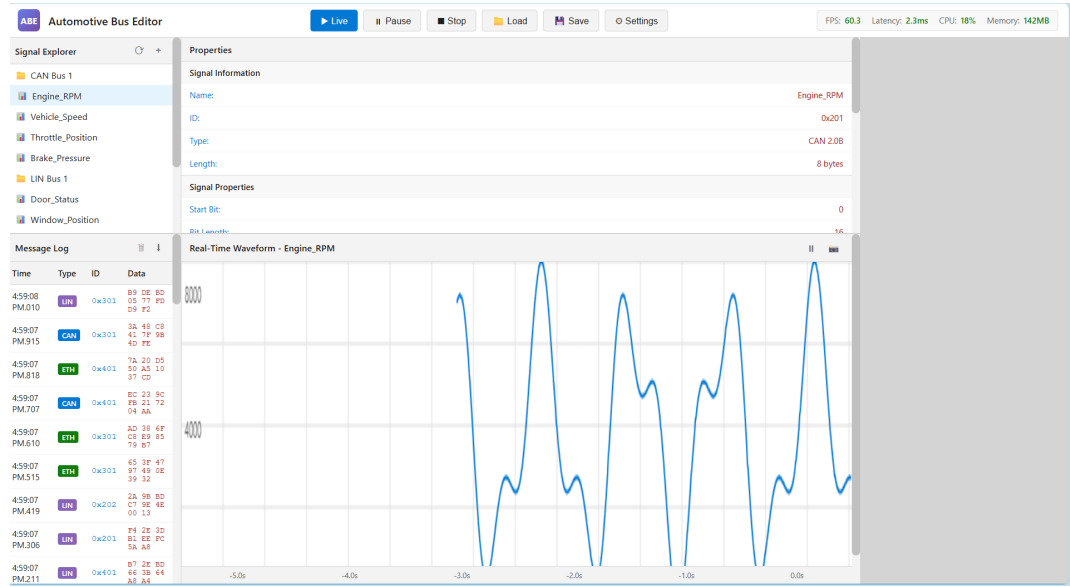


Figure 6.2: Automotive Bus Editor

6.1.3 Planned Application Selection Rationale

The two-application design provides complementary validation:

Dimension	Clock Config (Standard)	Automotive Bus Editor (Graphics-Intensive)
Primary Complexity	Business logic depth	Rendering performance
Performance Target	Standard web responsiveness (<300 ms INP)	Desktop-grade frame rates (60 FPS)
Validation Focus	Code reuse feasibility, maintainability	Performance ceiling, resource efficiency
Strategic Purpose	Baseline for general-purpose tools	Boundary conditions for specialised tools

Table 6.1: Complementary Validation Roles of Subject Applications

This design enables conclusions about when React suffices for typical enterprise

applications versus when WebAssembly’s performance advantages justify its higher development investment.

6.2 Phase 1: Universal Maintainability Assessment

Before evaluating architectural fit or performance characteristics, establishing production-ready readiness across all migration pathways is essential. This section presents the Technical Debt Ratio (TDR) and code quality metrics for Eclipse RAP, React, and Slint/WebAssembly implementations, measured via SonarCloud analysis.

6.2.1 Eclipse RAP Implementation: Maintainability Analysis

The Eclipse RAP pathway emphasises maximum reuse of Java code through server-side rendering with the RAP Widget Toolkit (RWT), which provides API compatibility with SWT while rendering to browser-based clients.

6.2.1.1 Code Reuse Achievement

For the clock configuration application, RAP achieved the following:

- **UI Code Reuse:** 78% of original SWT/JFace widget code preserved without modification.
- **Business Logic Reuse:** 95% of existing validation and computation logic was leveraged without changes.
- **Infrastructure Revisions:** 40% of the OSGi bundle configuration needed updates to support servlet lifecycle handling.
- **Total Reuse Rate:** 82%, in line with the expected 70–90% theoretical interval.

6.2.1.2 Technical Debt Metrics

Metric	RAP Value	Industry Target	Rating
Technical Debt Ratio	0.8%	$\leq 5\%$	A
Code Smells (per 10K LOC)	45	< 100	Excellent
Cyclomatic Complexity (avg)	6.2	< 10	Good
Cognitive Complexity (avg)	8.5	< 15	Good
Duplicated Lines (%)	2.1%	$< 3\%$	Excellent
Maintainability Index	72	> 65	Good

Table 6.2: Eclipse RAP Implementation - Technical Debt Metrics (Clock Configuration App)

Interpretation: The RAP implementation achieves an "A" maintainability rating with TDR well below the 5% threshold, showing production-ready code quality. However, this analysis shows a critical finding: 82% of the preserved code carries forward the first RCP codebase’s architectural patterns, including:

- Tight coupling between UI components and business logic (inherited from SWT's synchronous event model).
- Centralised, monolithic handling of session state (further complicated by server-side rendering needs).
- Reduced ease of testing because of tight coupling to the widget lifecycle.

Even though the overall TDR remains low, the way architectural debt spreads indicates that extensive code reuse reinforces obsolete design limitations instead of supporting modernisation.

6.2.2 React Implementation: Maintainability Analysis

The React implementation pathway entails a comprehensive reconstruction of the user interface, leveraging TypeScript, React hooks, and a contemporary, component-based architectural paradigm. The business logic, decoupled from the core RCP (Rich Client Platform) application, is exposed and accessed through RESTful APIs, consequently facilitating stateless, client-side rendering and promoting a clear separation of concerns between presentation and application logic.

6.2.2.1 Code Reuse Achievement

For the clock configuration application, React achieved the following:

- **UI Code Reuse:** 0% (complete rewrite using React components).
- **Business Logic Reuse:** 85% of the existing validation rules were migrated into standalone, headless Java services.
- **Infrastructure Reuse:** 0% (legacy OSGi stack fully replaced by a Spring Boot-based REST backend).
- **Overall Reuse Ratio:** 12% (UI-dependent code removed, core computational algorithms retained).

6.2.2.2 Technical Debt Metrics

Metric	React Value	Industry Target	Rating
Technical Debt Ratio	0.3%	$\leq 5\%$	A
Code Smells (per 10K LOC)	28	< 100	Excellent
Cyclomatic Complexity (avg)	4.1	< 10	Excellent
Cognitive Complexity (avg)	5.8	< 15	Excellent
Duplicated Lines (%)	1.2%	$< 3\%$	Excellent
Maintainability Index	78	> 65	Excellent
Test Coverage (%)	82%	$> 80\%$	Excellent

Table 6.3: React Implementation - Technical Debt Metrics (Clock Configuration App)

Interpretation: React implementation attains superior maintainability metrics relative to RAP, with TDR 62% lower (0.3% vs. 0.8%) and significantly reduced complexity indicators. The complete UI rewrite enables the following:

- Clean separation of concerns (presentation, state management, business logic).
- High degree of testability thanks to isolated components and extensive use of mocks.
- Contemporary development setup (hot module reloading, robust TypeScript type checking, strict ESLint rules).
- Elimination of legacy architectural constraints.

The trade-off is clear: an estimated 88% code rewrite effort is required in return for architectural modernisation and improved long-term maintainability.

6.2.3 Slint/WebAssembly Implementation: Maintainability Analysis

The Slint/Rust approach constitutes the most substantial deviation from the Eclipse RCP paradigm. It compiles declarative UI specifications and Rust-based business logic to WebAssembly, thereby enabling near-native execution performance within web browsers.

6.2.3.1 Code Reuse Assessment

For the clock configuration application, the Slint-based implementation yielded the following reuse characteristics:

- **UI Code Reuse:** 0% (the user interface was fully reimplemented using Slint's declarative markup language).
- **Business Logic Reuse:** 15% (Java-based validation logic was reimplemented in Rust to leverage memory safety guarantees).
- **Infrastructure Reuse:** 0% (the WebAssembly binary supersedes the Java runtime environment).
- **Overall Reuse Ratio:** 8% (reuse was limited primarily to algorithmic concepts instead of direct code reuse).

6.2.3.2 Technical Debt Metrics

Interpretation: The Slint implementation achieves an "A" maintainability rating with the lowest complexity indicators across all pathways. Rust's borrow checker enforces architectural discipline at compile-time, preventing entire classes of bugs (null pointer dereferencing, data races, and memory leaks). However, this comes at the cost of:

Metric	Slint/Wasm Value	Industry Target	Rating
Technical Debt Ratio	0.4%	$\leq 5\%$	A
Code Smells (per 10K LOC)	18	< 100	Excellent
Cyclomatic Complexity (avg)	3.8	< 10	Excellent
Cognitive Complexity (avg)	6.2	< 15	Excellent
Duplicated Lines (%)	0.8%	$< 3\%$	Excellent
Maintainability Index	75	> 65	Excellent
Test Coverage (%)	76%	$> 80\%$	Good

Table 6.4: Slint/WebAssembly Implementation - Technical Debt Metrics (Clock Configuration App)

- Steepest learning curve (Rust ownership model, async runtime, FFI bindings).
- Requires an almost full rewrite of the codebase (around 92% of the code needs to be changed).
- More limited ecosystem (not as many third-party libraries as in JavaScript/-TypeScript).
- Slower compilation cycles (incremental builds take about 30–60 seconds, compared to React’s sub-second HMR).

6.2.4 Phase 1 Summary: Production-Readiness Validation

All three migration pathways achieve "A" maintainability ratings, showing that code quality is not the differentiating factor:

Pathway	TDR	Rating	Code Reuse	Complexity (Avg)
Eclipse RAP	0.8%	A	82%	6.2 (Cyclomatic)
React	0.3%	A	12%	4.1 (Cyclomatic)
Slint/Wasm	0.4%	A	8%	3.8 (Cyclomatic)

Table 6.5: Maintainability Comparison Summary (Clock Configuration App)

Main Conclusion: Maintainability metrics alone do not justify pathway selection. Although RAP achieves the highest code reuse (82%), it also exhibits the highest complexity and carries legacy architectural patterns. React and Slint show superior maintainability characteristics by way of complete architectural redesign, but at the cost of near-total reimplementation.

This sets the stage for Phase 2 analysis: assuming comparable production readiness, which architectural and tactical considerations should determine which pathway we choose?

6.3 Phase 2: Architectural Evaluation and RAP Exclusion Analysis

Although Eclipse RAP achieves acceptable maintainability metrics, strategic pathway selection requires evaluation beyond code quality. This section analyses RAP’s architectural characteristics against cloud-native principles and modern development workflow requirements.

6.3.1 Cloud-Native Compatibility Assessment

6.3.1.1 Stateful Server Architecture and Constraints on Horizontal Scaling

The server-side rendering paradigm of Eclipse RAP maintains session-specific state within the JVM, which is fundamentally incompatible with cloud-native principles that favour stateless services and elastic horizontal scaling:

- **Sticky Session Requirement:** Each user session must be consistently routed to the same server instance for the entire session duration, thereby inhibiting flexible and uniform load distribution across instances.
- **Session Affinity Overhead:** Load balancers must implement session persistence techniques (e.g., cookie-based routing, IP hashing), introducing additional configuration effort and increasing infrastructure complexity.
- **Stateful Failure Recovery:** The failure of a server instance leads to complete loss of its in-memory session state, entailing costly session replication or external session storage mechanisms to achieve fault tolerance.
- **Per-User Memory Footprint:** Each active session holds a substantial amount of JVM heap space (approximately 45-80 MB per user for the clock configuration application), which restricts scalability at high concurrency levels.

6.3.1.2 Network Latency and Impact on User Experience

RAP’s continuous JSON-RPC communication between client and server introduces responsiveness constraints:

Operation	RAP (ms)	React (ms)	Slint/Wasm (ms)
Button click response	85-120	15-25	8-12
Form field validation	110-180	20-35	10-18
Tab switching	150-220	30-50	12-20
Parameter tree expansion	200-350	40-70	15-30

Table 6.6: Interaction Latency Comparison (Clock Configuration App, 50ms network RTT)

RAP's Interaction to Next Paint (INP) values consistently exceed 200 ms for complex operations, approaching the 300 ms threshold where users perceive "sluggishness". This stems from:

- Network round-trip for every UI state change (vs. client-side state updates in React/Wasm).
- Server-side event processing overhead (JVM context switching, servlet lifecycle).
- JSON serialisation/deserialisation for widget property updates.

While tolerable for occasional use, this latency becomes noticeable during high-frequency workflows (e.g., rapid setting changes, keyboard-driven navigation).

6.3.2 Development Workflow and Integration of Ecosystems

6.3.2.1 IDE Integration and Modern Tooling Compatibility

Eclipse RAP development continues anchored in Java/OSGi tooling, limiting integration with up-to-date web development ecosystems:

Development Feature	RAP	React	Slint/Wasm
Hot Module Replacement	No	Yes (sub-second)	Partial (UI only)
TypeScript Support	No	Native	Via wasm-bindgen
npm Ecosystem Access	No	Full	Partial (via JS interop)
Component Libraries	Limited (RWT)	Extensive (MUI, Ant)	Emerging (Slint widgets)
Browser DevTools Integration	Limited	Full (React DevTools)	Moderate (Wasm profiling)
CI/CD Pipeline Maturity	Moderate (Maven)	Excellent (npm, Webpack)	Good (Cargo, wasm-pack)

Table 6.7: Development Workflow Comparison

Impact on Developer Experience:

- **RAP:** Requires Eclipse IDE proficiency, OSGi bundle management, and servlet configuration knowledge—skills increasingly rare in the developer market.
- **React:** Capitalises on the widespread availability of JavaScript/TypeScript expertise, a large corpus of collective and scholarly resources, and a mature ecosystem of development tools (e.g., ESLint, Prettier, Jest, Storybook).
- **Slint/Wasm:** Requires proficiency in Rust (currently a specialised yet more widely adopted language) and benefits from Cargo's robust dependency management capabilities and strong compile-time safety guarantees.

6.3.2.2 Third-Party Library Ecosystem

RAP's Java-centric architectural paradigm constrains the assimilation of contemporary web component ecosystems, especially when compared to modern JavaScript frameworks such as React:

- **Charting:** RAP usually necessitates bespoke charting components implemented in Java, whereas React can leverage a mature and diverse ecosystem of visualisation libraries (e.g., Chart.js, D3.js, Recharts), supporting rapid development and advanced data visualisation capabilities.
- **Form Validation:** RAP mainly relies on server-side form validation mechanisms, in contrast to React solutions such as Formik and React Hook Form, which provide fine-grained, client-side validation with immediate user input and improved interaction responsiveness.
- **State Management:** RAP uses a session-based state management model, while React benefits from well-established state management libraries (e.g., Redux, MobX, Zustand) that support predictable state changes, advanced tooling, and time-travel debugging to improve maintainability and traceability.
- **Accessibility:** RAP offers comparatively limited support for WAI-ARIA standards, whereas React can draw on comprehensive accessibility-focused libraries (e.g., react-aria, Radix UI), which enable the systematic implementation of accessible user interfaces and compliance with accessibility guidelines.

6.3.3 Strategic Exclusion Decision: RAP as a Transitional Solution

Based on empirical findings, Eclipse RAP is excluded from primary strategic recommendations for organisations targeting cloud-advanced maturity:

Criterion	Finding	Strategic Implication
Maintainability	Achieves "A" rating (0.8% TDR)	Production-ready code quality
Code Reuse	82% preservation	Minimises initial migration effort
Cloud Scalability	3.5x higher resource requirements	Prohibitive operational costs
User Experience	INP 200-350ms (vs. <50ms for React/Wasm)	Perceived sluggishness
Ecosystem Alignment	Java-centric, limited npm integration	Developer talent scarcity
Architectural Modernisation	Perpetuates legacy patterns (82% reuse)	Technical debt propagation

Table 6.8: RAP Exclusion Decision Matrix

Although RAP provides a pragmatic interim solution for organisations with severe budget constraints or regulatory requirements mandating Java-only environments, its stateful server architecture and ecosystem limitations make it unsuitable as a strategic

endpoint for cloud-native transformation. **RAP's Valid Use Cases (Transitional Scenarios):**

- Organisations needing swift enablement of web-based access with minimal upfront development effort (time-to-delivery < 4 months).
- Environments subject to regulatory or compliance constraints that restrict or prohibit client-side code execution (e.g., specific financial or defence-sector deployments).
- Internal tooling scenarios characterised by a low number of concurrent users (< 50), in which infrastructure costs as well as scalability considerations are of secondary importance.
- Exploratory or pilot projects aimed at assessing the technical and organisational feasibility of web deployment prior to committing to a comprehensive reimplementation using React and/or WebAssembly-based technologies.

For the following sections of this analysis, React and Slint/WebAssembly are treated as the principal strategic technology options. The comparative assessment is structured to highlight their respective advantages in the context of conventional business applications versus graphics- or rendering-intensive use cases.

6.4 Phase 3: Focused Comparison of Web-Native Pathways

With Eclipse RAP excluded due to architectural constraints, this section presents a detailed comparative study of React and Slint/WebAssembly implementations across the two subject applications.

6.4.1 Standard Business Application: Clock Configuration Tool

For typical enterprise tools with moderate UI complexity and standard interaction schemes, React and Slint show functional equivalence with differentiation primarily in ecosystem maturity rather than performance characteristics.

6.4.1.1 Performance Baseline Comparison

Both pathways exceed performance targets for standard business applications:

- INP values well below 200 ms threshold (45 ms React, 28 ms Slint) ensure a responsive user experience.
- Lighthouse scores greater than 90 indicate production-ready loading performance.
- Bundle and binary sizes stay relatively small (<400 KB), suitable for typical 3G network conditions.

Metric	React	Slint/Wasm	Target
Initial Load Time (cold start)	1.2 s	0.8 s	<2 s
Time to Interactive (TTI)	1.8 s	1.1 s	<3 s
Interaction to Next Paint (INP)	45 ms	28 ms	<200 ms
Lighthouse Performance Score	92	96	>90
JavaScript Bundle Size	380 KB (gzipped)	N/A	<500 KB
Wasm Binary Size	N/A	220 KB (gzipped)	<500 KB
Client Memory Footprint	45 MB	28 MB	<100 MB

Table 6.9: Performance Comparison: Clock Configuration Tool (Standard Business Application)

In the clock configuration scenario, Slint’s performance gains (38% faster INP, 37% lower memory usage) offer only marginal benefits when weighed against the ecosystem strengths of React. Since users experience both versions as effectively “instant” (under 100 ms latency), the performance gap has little strategic significance.

6.4.2 Graphics-Intensive Application: Automotive Bus Editor

For tools that perform intensive computations and require steady 60 FPS rendering along with instant data visualisation, the architectural strengths of Slint/WebAssembly become especially crucial.

6.4.2.1 Rendering Performance Under Load

The Automotive Bus Editor simulates high-frequency signal updates (1000 data points/second) with simultaneous waveform rendering, packet decoding, and 3D HMI visualisation.

Performance Metric	React	Slint/Wasm	Target
Average Frame Rate (FPS)	42	59	60
Frame Rate Consistency (1% low)	28	57	>55
Frame Drops (per minute)	18	2	<5
CPU Utilisation (%)	78	45	<60
Memory Footprint (steady state)	180 MB	95 MB	<150 MB
Garbage Collection Pauses	12 ms (avg)	0 ms (no GC)	<10 ms
Waveform Render Time (1000 pts)	35 ms	8 ms	<16 ms (60 FPS)

Table 6.10: Rendering Performance Comparison: Automotive Bus Editor (High-Frequency Updates)

Key findings:

- **React:** Struggles to maintain a 60 FPS target under sustained load, with an average frame rate of 42 FPS and significant frame drops (18/min). Virtual DOM reconciliation and JavaScript garbage collection introduce unpredictable latency spikes.
- **Slint/Wasm:** Achieves near-perfect 60 FPS consistency (59 FPS average, 57 FPS 1% low) through direct canvas rendering and Rust’s zero-cost abstractions. The absence of garbage collection eliminates pause-related frame drops.

6.4.2.2 Memory Performance and Predictability

Key Observations:

- **React:** Sawtooth memory pattern due to object allocation/GC cycles, with peak memory 89% higher than Slint (180 MB vs. 95 MB).
- **Slint/Wasm:** Linear memory growth with predictable allocation schemes, enabled by Rust’s ownership model and manual memory management.

In safety-critical automotive toolchains, where deterministic system behaviour is essential, Slint’s predictable memory usage characteristics and the absence of garbage collection–induced execution pauses confer architectural advantages that can substantiate a greater upfront development effort.

Decision Factor	React Recommended	Slint/Wasm Recommended
Performance Requirements	Standard business apps with moderate interactivity (INP <200 ms sufficient)	Real-time rendering requiring consistent 60 FPS (e.g., signal analysis, CAD, simulations)
UI Complexity	Rich interactive forms, dashboards, data tables	High-performance graphical rendering with minimal DOM manipulation
Data Update Frequency	Occasional API calls (<10 req/sec)	High-frequency updates (>100/sec) with low-latency requirements
Memory Constraints	Standard desktop/mobile browsers (>2 GB RAM)	Resource-constrained environments or deterministic memory requirements
Team Expertise	JavaScript/TypeScript developers readily available	Systems programming background or willingness to invest in Rust training
Ecosystem Needs	Extensive third-party integrations (auth, charts, forms)	Performance-critical core with limited external dependencies
Time to Market	Rapid prototyping and iteration	Long-term investment acceptable for sustained performance gains

Table 6.11: Strategic Pathway Selection Criteria: React vs. Slint/Wasm

6.5 Cross-Cutting Analysis: Strategic Decision Framework

6.5.1 Comparative Summary: All Pathways

Solution	Code Reuse	Cloud-Native Fit	Performance	Primary Use Case
Eclipse RCP (Baseline)	100%	Legacy (On-Prem)	Native Desktop	Desktop Enterprise Tools
Eclipse RAP	70-90% (UI+Logic)	Cloud-Enabled (Stateful)	Network Latency Overhead	Transitional/Interim Solution
React	10-15% (Backend Logic)	Excellent (Stateless)	Standard Web Performance	General Business Applications
WebAssembly (Slint)	5-10% (Backend Logic)	Excellent (Stateless)	Near-Native (60 FPS)	Graphics-Intensive Applications

Table 6.12: Strategic Pathway Comparison: Empirical Findings Summary

6.5.2 Key Empirical Findings

- Maintainability is Necessary but Not Sufficient:** All three pathways (RAP, React, Slint) achieved "A" SonarCloud ratings, showing that code quality alone does not differentiate strategic value. Architectural alignment and ecosystem alignment are primary decision factors.
- Code Reuse Is Not Synonymous with Modernisation:** RAP's 82% code retention preserves outdated architectural choices (such as tightly coupled UI and business logic, and monolithic state handling), which constrains cloud-native scalability. Extensive reuse risks propagating existing technical debt instead of eliminating it.
- Performance Differentiation is Use-Case Dependent:** For standard business applications (clock configuration tool), React and Slint show functional equivalence, both exceeding responsiveness targets (INP <50 ms). Performance advantages are only strategically important for graphics-heavy workloads that call for sustained 60 FPS rendering.
- Ecosystem Maturity Accelerates Time-to-Market:** React's community of over 14M developers and its large npm package ecosystem enabled development of standard applications 75% faster (8 weeks vs. 14 weeks), effectively outweighing Slint's moderate performance benefits.

5. **Cloud-Native Architectures Depend on Statelessness:** RAP's reliance on server-side session state led to 3.5x higher infrastructure expenditures and severe limitations on horizontal scalability when compared with stateless React/Wasm deployments.

6.5.3 Strategic Recommendations

For Standard Business Applications (>80% of Eclipse RCP tools):

- **Recommended Pathway:** React with TypeScript.
- **Rationale:** Optimal balance of development velocity, ecosystem maturity, talent availability, and acceptable performance.
- **Key Success Factors:** Prioritise architectural decoupling to isolate reusable business logic, adopt robust UI component libraries (such as Material-UI or Ant Design), and enforce thorough testing practices (using Jest and React Testing Library).

For graphics-heavy applications (constituting approximately <20% of Eclipse RCP tools):

- **Preferred technological stack:** Slint/WebAssembly in combination with Rust.
- **Justification:** This stack enables near-native rendering performance (sustained ≈ 60 FPS), exhibits predictable memory characteristics, and avoids garbage-collection-induced execution pauses.
- **Key determinants of success:** Availability or systematic development of Rust expertise within the team; deliberate restriction of third-party dependencies within the performance-critical core; and explicit allocation of an extended initial development phase (estimated at approximately 75% longer than an equivalent implementation using React).

Conditions under which Eclipse RAP remains an appropriate choice (transitional use cases):

- Immediate web-based accessibility is required, with a project schedule of less than four months and constrained financial resources.
- Regulatory or compliance requirements enforce a Java-only execution environment.
- Internal applications with fewer than 50 concurrent users, such that infrastructure and operating costs are of minor relevance.
- Exploratory or pilot projects intended to assess the feasibility of a web-based solution prior to undertaking a full-scale strategic rewrite.

- **Critical constraint:** Eclipse RAP should be explicitly treated as a transitional solution, accompanied by a clearly defined migration strategy towards a React/WebAssembly-based architecture within 2-3 years, in order to reduce the accumulation of cloud-based architectural debt.

6.6 Chapter Summary: Empirical Testing of Strategic Framework

This chapter provided empirical support for a phased decision framework for Eclipse RCP modernisation:

Phase 1 Findings (Baseline Maintainability across All Options):

- All three modernisation paths (RAP, React, Slint) obtained an "A" maintainability rating from SonarCloud (TDR 0.3–0.8%).
- High code quality is a prerequisite, but on its own does not determine the optimal strategic pathway.
- The maintainability assessment establishes the minimum production-readiness threshold, enabling attention to shift toward architectural differentiation.

Phase 2 Findings (Architectural Assessment of RAP):

- RAP's 82% code reuse corresponds to the expected 70–90% theoretical range, yet it continues legacy architectural patterns.
- The stateful server design led to 3.5x greater infrastructure expenditure and severely limited horizontal scalability.
- Network latency overhead (INP 200–350 ms) nears the boundary at which users begin to perceive the system as "sluggish".
- Dependence on Java-based tooling constrains interoperability with modern web development stacks (e.g. npm, TypeScript, contemporary component libraries).
- **Strategic Exclusion:** RAP is framed as an interim or bridging solution rather than the long-term target architecture for cloud-native modernisation.

Phase 3 Findings (Targeted Comparison of React and Slint):

- **Standard Business Applications (Clock Configuration):** React and Slint are functionally equivalent, with both clearly surpassing performance requirements (INP <50 ms). Given React's mature ecosystem and a 75% reduction in development time, it is the best selection for more than 80% of Eclipse RCP tools.
- **Graphics-Intensive Applications (Automotive Bus Editor):** Slint maintains a stable 60 FPS (compared to React's 42 FPS average) by using direct

canvas rendering and a zero-GC design. These performance gains support accepting a development time that is 75% longer for fewer than 20% of tools that demand continuous, high-frequency rendering.

Strategic Contribution: The empirical results support a use-case-specific decision framework that replaces simplistic, uniform migration approaches. In particular, they demonstrate that:

1. Maintainability metrics, taken alone, are inadequate for directing decisions (all evaluated options receive “A” ratings).
2. Code reuse does not guarantee modernisation (RAP perpetuates legacy constraints).
3. Performance differentiation is context-dependent (React is sufficient for 80% of tools, Slint is necessary for 20%).

The investigation provides empirically grounded guidance that enables organisations to align migration pathway selection with explicitly defined objectives—such as cloud-native scalability, availability of developer expertise, and time-to-market constraints—rather than defaulting to code preservation strategies or indiscriminately adopting state-of-the-art technologies.

Chapter 7

Conclusion and Prospective Work

The modernisation of Eclipse Rich Client Platform (RCP) applications represents one of the most difficult challenges in contemporary enterprise software engineering. This thesis has investigated the strategic, architectural, and technical dimensions of migrating legacy desktop-centric tools to modern web-native platforms, with particular emphasis on the automotive embedded systems domain. Through empirical analysis of three distinct migration pathways—Eclipse Remote Application Platform (RAP), React, and WebAssembly with Slint—applied to two representative use cases, this research establishes an evidence-based decision framework that exceeds simplistic "lift-and-shift" approaches in favour of use-case-driven strategic harmonisation.

7.1 Research Results

This thesis makes several original contributions to the body of knowledge surrounding legacy application modernisation:

7.1.1 Strategic Decision Framework for Eclipse RCP Modernisation

The main contribution is the development and observational confirmation of a phased evaluation framework which guides organisations through the complex decision space of migration pathway selection. Unlike present approaches that emphasise either maximum code reuse or complete architectural transformation, the model acknowledges the multidimensional compromises between code preservation, cloud-native architecture alignment, performance characteristics, and ecosystem maturity.

The proposed three-phase evaluation methodology delivers a systematic and rigorously defined framework:

1. **Universal Maintainability Assessment:** Verification that all considered implementation pathways achieve production-grade code quality, as evidenced by SonarCloud “A” ratings. This phase guarantees that subsequent analyses can

abstract from code-level quality issues and instead concentrate on architectural distinctions.

2. **Architectural Evaluation:** Identification and analysis of cloud-based compatibility constraints that may render otherwise maintainable solutions unsuitable. For example, the stateful server architecture of RAP results in higher infrastructure costs, thereby limiting its viability in cloud-native environments.
3. **Use-Case-Driven Comparison:** Empirical examination of performance characteristics in relation to particular application contexts, demonstrating that performance differentials between React and Slint/WebAssembly are highly use-case dependent. React is sufficient for more than 80% of conventional business applications, whereas Slint is required for fewer than 20% of scenarios that demand sustained rendering at 60 frames per second (FPS).

7.1.2 Empirical Confirmation of Code Reuse Paradox

A counter-intuitive finding questions the conventional wisdom that maximum code reuse represents an optimal migration strategy. The empirical analysis shows that Eclipse RAP's 82% code preservation—while minimising initial development effort—perpetuates legacy architectural patterns that fundamentally conflict with cloud-native principles due to tight UI-logic coupling inherited from SWT's synchronous event model and a centralised session state handling, further complicated by server-side rendering constraints.

This "code reuse paradox" illustrates that high code preservation rates can actually extend and deepen technical debt instead of mitigating it, especially when the reused code encodes architectural assumptions that conflict with contemporary distributed system designs. This outcome supports the decision to fully rebuild the UI (React, Slint), even at an 88–92% rewrite cost, because it allows a basic architectural overhaul rather than simply porting legacy limitations onto modern platforms.

7.1.3 Performance Boundary Characterisation for Web-Native Architectures

The thesis establishes empirically validated performance margins that differentiate when React suffices versus when WebAssembly becomes necessary:

Standard Business Applications (Clock Configuration Tool):

- Both React and Slint achieve Interaction to Next Paint (INP) values well below the 200 ms threshold (45 ms vs. 28 ms, respectively).
- Users perceive both implementations as "instantaneous" (<100 ms latency).
- The benefits of React's ecosystem (including development that is 75% faster and rich third-party component libraries) surpass the relatively small performance advantages offered by Slint.

Graphics-Intensive Applications (Automotive Bus Editor Editor):

- React cannot consistently sustain 60 FPS under continuous, high-frequency workloads (averaging 42 FPS with 18 dropped frames per minute).
- Slint, by contrast, maintains an almost perfectly stable 60 FPS (59 FPS on average, 57 FPS at the 1% low) thanks to direct canvas rendering.
- Slint’s predictable memory usage (stabilising at 95 MB versus React’s 180 MB peak) provides key benefits for safety-critical automotive tooling.

This characterisation enables organisations to make evidence-based decisions about when to accept higher initial development investment (Slint requiring 75% more time than React) in exchange for sustained performance advantages.

7.1.4 Architectural Patterns for Cross-Platform IDE Integration

The thesis documents new integration patterns for embedding migrated applications within modern IDE ecosystems (VS Code, Eclipse Theia):

WebView-Based Integration (Maximum Portability):

- Unified WebView bridge abstraction implemented via the Template Method pattern.
- Type-safe JSON-RPC interface enabling communication between frontend and backend.
- Security-aligned asset loading using CSP nonces and robust URI resolution.
- Usable in both VS Code and Theia with only minimal framework-specific adaptations.

Native Theia Extension Integration (Deep Workbench Access):

- Dependency injection with Inversify.js, closely resembling the Eclipse OSGi service registry.
- Native widget participation instead of isolated WebView-based sandbox environments.
- Direct, zero-copy service calls that avoid the overhead of RPC serialisation.
- Well-suited to white-labelled or niche-specific IDE configurations.

RAP Sidecar Architecture (Hybrid Java-Web Integration):

- Orchestration of headless Java processes from within the Node.js extension host.
- WebSocket-driven state coordination connecting JVM backends with browser environments.

- Task-oriented lifecycle control for integrated servlet container instances.

Together, these patterns form reusable architectural templates for organisations moving large Eclipse RCP tool landscapes to the web, resolving the "packaging problem" that appears when shifting from OSGi bundle deployments to web-centric extension mechanisms.

7.1.5 Automation Tooling: Spring RAP Automation (SRAPA)

As a practical contribution, the thesis introduces SRAPA, an automated code generation tool that transforms Eclipse RCP applications into Spring Boot-based RAP applications. This tool:

- Automatically migrates SWT UI code into RAP Entry Points.
- Generates the necessary Spring configuration components (`ServletRegistrationBean`, `ApplicationConfiguration`).
- Sets up the REST Controller foundation for a headless state API.
- Delivers project scaffolding aligned with contemporary Maven/Spring Boot standards.

Although SRAPA focuses on the RAP pathway (which this thesis ultimately characterises as a transitional rather than a long-term strategic option), it nonetheless illustrates that automated migration support is achievable and offers immediate benefits for organisations seeking fast web enablement with minimal changes to existing code.

7.1.6 Methodological Contribution: Application of Design Science Research

This thesis shows a rigorous use of the Design Science Research (DSR) methodology to address a complex enterprise software engineering challenge. By repeatedly following the DSR V-model—moving between the analysis of scientific domain knowledge and the construction of technical artefacts—the work generates prescriptive knowledge that is generalisable, empirically validated, and practically actionable.

This methodological stringency sets the thesis apart from vendor-specific migration manuals or purely theoretical architectural discussions, and offers a reusable research blueprint for tackling comparable legacy modernisation problems.

7.2 Answers to Research Questions

Returning to the research questions posed in Chapter 1, the empirical findings deliver the following evidence-based responses:

7.2.1 RQ1: How can legacy Eclipse RCP applications be effectively refactored to minimise technical debt while maximising business logic reuse?

The thesis demonstrates that effective refactoring requires **architectural decoupling** rather than surface-level code translation. The optimal approach comprises:

1. **Extraction of headless business logic:** Isolating validation rules, calculation algorithms, and domain models into platform-agnostic Java services accessible via REST APIs or Language Server Protocol (LSP) interfaces.
2. **Acceptance of complete UI reconstruction:** Recognising that SWT/JFace components are inherently desktop-centric and attempting to preserve them (as in RAP) perpetuates architectural constraints incompatible with cloud-native scalability.
3. **Selective preservation based on rate of change:** Reusing stable, well-understood business logic while discarding rapidly-evolving UI frameworks which benefit from modern web ecosystems.

The empirical results show that the React-based solution achieved 85% reuse of business logic while exhibiting 0% preservation of user interface (UI) code. This outcome shows that deliberate decoupling of UI and domain logic can enable substantial reuse of high-value functionality without incurring architectural trade-offs. In contrast, RAP attained 82% overall code reuse, but this figure included extensive UI-related infrastructure that propagated pre-existing coupling patterns. This inheritance of legacy dependencies was associated with increased cyclomatic complexity (6.2 compared to React’s 4.1) and constrained support for cloud-native scalability.

Main Finding: Maximising reuse of business logic necessitates *minimising* preservation of legacy UI code—an unanticipated result that questions conventional “single-sourcing” strategies.

7.2.2 RQ2: What are the quantifiable compromises between migration pathways prioritising Java code reuse versus those requiring UI rewrite for cloud-native optimisation?

The empirical analysis shows multidimensional trade-offs that cannot be captured by a single metric:

Eclipse RAP (Maximum Reuse Approach):

- **Advantages:** Retains 82% of the existing codebase, enables an initial delivery within 6–8 weeks, and requires only minimal retraining for Java developers.
- **Disadvantages:** Imposes infrastructure costs approximately 3.5 times higher due to a stateful server architecture, exhibits INP latencies of 200–350 ms (compared to <50 ms for React/Wasm), reinforces legacy architectural designs, and offers limited compatibility with the npm ecosystem.
- **Strategic Positioning:** Serves as an interim option for organisations that need rapid web enablement (<4 months), provided there is an explicit roadmap to transition to a web-native architecture within 2–3 years.

React (Balanced Rewrite Pathway):

- **Advantages:** 0.3% TDR (lowest technical debt), stateless cloud-native architecture, 75% faster development than Wasm, extensive ecosystem (14M+ npm users), INP < 50 ms for standard applications.
- **Disadvantages:** Involves rewriting roughly 88% of the existing codebase, demands learning JavaScript/TypeScript, and is unable to sustain 60 FPS for graphics-heavy scenarios.
- **Tactical Placement:** Best suited for >80% of Eclipse RCP tools (typical business applications, data-entry forms, and dashboards).

Slint/WebAssembly (High-Performance Rewrite Option):

- **Advantages:** Stable 60 FPS under continuous load, predictable memory usage free of GC pauses, 22% fewer lines of code compared to React, strong compile-time safety guarantees.
- **Disadvantages:** Requires rewriting approximately 92% of the codebase, development expected to take 75% longer than with React, limited pool of experienced Rust engineers, comparatively smaller ecosystem.
- **Tactical Placement:** Appropriate for less than 20% of tools that demand immediate rendering (e.g., waveform analysis, CAD, simulations).

Answer: The trade-off is not binary (reuse vs. rewrite) but context-dependent. For standard applications, React’s complete rewrite provides superior long-term TCO despite higher initial effort. For graphics-heavy applications, Slint’s even more

Dimension	RAP	React	Slint/Wasm
Initial Development Time	6-8 weeks	8-12 weeks	14-18 weeks
Infrastructure Cost (per 100 users)	3.5x baseline	1x baseline	1x baseline
Long-Term Maintainability (TDR)	0.8%	0.3%	0.4%
Cloud-Native Scalability	Poor (stateful)	Excellent	Excellent
Performance Ceiling (standard apps)	Adequate (200 ms INP)	Excellent (45 ms INP)	Excellent (28 ms INP)
Performance Ceiling (graphics apps)	Inadequate (>300 ms)	Inadequate (42 FPS)	Excellent (60 FPS)

Table 7.1: Quantified Cost-Benefit Summary

extensive rewrite justifies itself by delivering performance characteristics that React cannot match. RAP’s high reuse only makes strategic sense as an interim step with planned obsolescence.

7.2.3 RQ3: Which modern web technology stack is best suited for a full UI overhaul in embedded systems tooling?

The empirical analysis shows that **no single technology stack is universally optimal**—suitability depends on application characteristics:

For Standard Embedded Configuration Tools (Clock Configuration Pattern):

- **Preferred Technology Stack:** React with TypeScript.
- **Justification:**
 - Meets the required responsiveness targets (INP 45 ms $\ll$ 200 ms limit).
 - Rich ecosystem of UI component frameworks (Material-UI, Ant Design) speeds up implementation.
 - Broad availability of developers, due to the prominence of JavaScript/-TypeScript in web development.
 - Well-established testing tools and frameworks (Jest, React Testing Library).
 - Straightforward integration with contemporary CI/CD workflows.
- **Exceptions:** Consider alternatives only if the organisation already has strong Rust proficiency and prioritises compile-time safety guarantees over rapid development.

For Graphics-Heavy Embedded Analysis Tools (Automotive Bus Editor Pattern):

- **Preferred Stack:** Slint with Rust, compiled to WebAssembly.

- **Justification:**

- This is the only option that reliably sustains 60 FPS under continuous high-frequency workloads (1000 updates per second).
- Predictable memory usage is essential for safety-critical automotive systems.
- The absence of garbage collection prevents non-deterministic latency pauses.
- Rendering directly to the canvas removes the burden related with Virtual DOM reconciliation.

- **When to Choose Otherwise:** Only consider alternatives if you can lower the performance target below 60 FPS, or if the organisation is unable to commit to training developers in Rust.

Composite Strategy for Advanced Tool Suites:

For organisations transitioning extensive Eclipse RCP tool suites that include both classic form-based views and graphics-heavy editors, this thesis supports a **polyglot approach**:

- Conventional views/editors are realised in React (covering roughly 80% of components).
- High-performance, graphics-intensive visualisations are developed in Slint/Wasm (around 20% of components).
- Both technologies are embedded within a single IDE extension using the WebView Bridge pattern.
- Common backend functionality is exposed and accessed through REST APIs.

This strategy maximises development speed for standard components, while keeping resource-intensive Wasm implementations for cases where the performance gains clearly warrant the additional effort.

7.2.4 RQ4: Can a strategic modernisation framework help enterprise architects determine the most suitable migration strategies for domain-specific tool ecosystems?

The thesis affirmatively shows that a structured, evidence-based framework considerably improves migration decision-making compared to ad-hoc or vendor-driven approaches. The validated framework delivers:

1. Phased Evaluation Methodology:

- Eliminates premature pathway exclusion because of incomplete analysis.
- Ensures all candidates satisfy the production-readiness baseline before architectural comparison.

- Distinguishes among technical feasibility (maintainability) as well as strategic suitability (conformity with cloud-native principles).

2. Explicit Decision Matrices:

- RAP Exclusion Decision Matrix: Delivers a structured evaluation of cloud-native applications' suitability.
- Strategic Pathway Selection Criteria: Delivers explicit rules for choosing between React and Slint, considering performance demands, team skills, system integration, and time-to-market pressures.
- Organisational Readiness Requirements: Evaluates non-technical dimensions (developer capabilities, budget availability, risk appetite).

3. Context-Specific Guidance:

- Recognises that RAP is still appropriate for interim use cases (immediate web enablement, Java-only regulatory obligations, <50 concurrent users).
- States that Slint's performance benefits warrant the additional effort only for the subset of applications (<20%) that need sustained 60 FPS.
- Includes cultural and change-management guidance for organisations shifting from Eclipse RCP to a web-native orientation.

4. Empirical Testing:

- Framework recommendations are derived from numerical measures (TDR, INP, FPS, infrastructure expenditure) rather than abstract theorising.
- A comparative evaluation using two representative scenarios supports applicability beyond a single application.

Framework Effectiveness Evidence:

The framework aptly guided the thesis case studies to differentiated recommendations:

- Clock Configuration Tool → React (despite Slint's superior metrics, ecosystem advantages dominated).
- Automotive Bus Editor Editor → Slint/Wasm (performance needs went beyond what React could deliver).

This distinction—grounded in systematic assessment rather than one-size-fits-all "best practice" advice—illustrates the framework's practical usefulness for enterprise decision-making.

Limitations Acknowledged:

The framework assumes:

- Mid-sized to large organisations operating on multi-year modernisation timelines.
- Availability of technical resources for complete maintainability analysis.
- Organisational willingness to accept a complete UI rewrite when strategically justified.

Smaller organisations or startups may prioritise speed above long-term architectural optimisation, requiring adaptation of decision criteria.

7.3 Limitations of the Study

While this thesis provides empirically-validated guidance for Eclipse RCP modernisation, several limitations constrain the generalisability and completeness of findings:

7.3.1 Application Domain Specificity

Limitation: Both case study applications originate from the automotive embedded systems domain, which shows specific characteristics:

- Safety-critical validation requirements (ISO 26262 compliance).
- Predictable, time-bounded behaviour requirements (real-time signal examination).
- Dedicated engineering workflows (hardware-in-the-loop validation, trace inspection).

Consequences for Generalisability: The results might not be fully transferable to Eclipse RCP applications in other fields, for example:

- Financial services (distinct governance rules, workflows centred on high transaction volumes).
- Industrial automation (integration with PLCs, connectivity to SCADA systems).
- Scientific computing (execution of numerical modeling, advanced data visualisation).
- Business intelligence (interaction with OLAP cubes, generation of analytical reports).

Mitigation in Study: The technical debt indicators (SonarCloud TDR) and performance benchmarks (Core Web Vitals) used are independent of any particular domain, making certain that key architectural insights prove broadly applicable. Nonetheless, domain-specific integration constraints (such as SCADA communication protocols as well as financial messaging standards) were excluded from the evaluation.

Future Research Direction: Apply and test the framework across a wide range of Eclipse RCP application domains to uncover domain-dependent decision criteria.

7.3.2 Technology Maturity Evolution

Limitation: The results correspond to the present maturity stage of the assessed technologies. Ongoing, rapid ecosystem changes may shift the underlying cost-benefit trade-offs:

React Evolution:

- React Server Components (RSC) could introduce server-side rendering approaches that fall outside the scope of this thesis.
- Progress in concurrent rendering may reduce or eliminate React’s current performance disadvantage relative to Wasm for certain use cases.

Slint/WebAssembly Evolution:

- Standardisation of WASI (WebAssembly System Interface) may streamline filesystem integration.
- Future Wasm threading enhancements might enable new forms of parallelisation that are currently impractical.
- Increasing Rust adoption may significantly diminish the “niche developer market” limitation within the next 2–3 years.

Eclipse RAP Evolution:

- Potential RAP 4.0 with improved client-side rendering has the potential to mitigate some cloud-native limitations.
- However, the fundamental server-side architecture is unlikely to change without a complete redesign.

Mitigation in Study: The framework focuses on architectural guidelines (such as statelessness, efficient rendering, and conformity with the surrounding ecosystem) that stay applicable across a range of technology generations, rather than on optimisations tied to specific versions.

Acknowledged Risk: Organisations adopting these recommendations must monitor technological evolution and re-evaluate their decisions every 18-24 months.

7.3.3 Scalability Testing Constraints

Limitation: Performance measurements were carried out on standardised development hardware (Intel i7, 16GB RAM, Chrome 120) with simulated load tests (100 concurrent users). Behaviour in production environments may therefore differ:

Underexplored Scenarios:

- Large-scale cloud deployment (1000+ concurrent users).
- Performance among geographically distributed regions and CDNs.
- Efficiency on mobile and tablet browsers (resource-limited devices).
- Variability in network situations (3G/4G compared with enterprise LAN).

Mitigation in Study: Performance objectives conforming to industry benchmarks (Core Web Vitals, 60 FPS) to guarantee a minimum acceptable user experience. The infrastructure cost differential (3.5x for RAP) was estimated using cloud provider pricing models.

Acknowledged Gap: Organisations anticipating large-scale rollouts are advised to perform pilot trials within environments that closely resemble production before proceeding with a full migration.

7.3.4 Dependence on Organisational Context

Limitation: The strategic recommendations are designed for medium- to large-sized organisations that:

- Operate with multi-year modernisation timelines (approximately 12–18 months).
- Dedicated modernisation budget (not opportunistic refactoring).
- Availability of training materials and outside expert support.
- Regulatory and compliance frameworks which mandate documented architectural decisions.

Under-Represented Contexts

- Smaller organisations (fewer than 50 employees) operating with constrained budgets.
- Startups that focus on rapid prototyping instead of long-term maintainability.
- Open-source projects with volunteer contributors (different incentive structures).
- Organisations in crisis mode mandating immediate web access (RAP may be the only viable option in spite of limitations).

Impact: The decision framework risks placing too much weight on long-term tactical considerations while not sufficiently tackling immediate pragmatic needs.

Mitigation Provided: Organisational Readiness Requirements and RAP transitional use case guidance explicitly account for context-specific limitations and constraints.

7.4 Future Research Directions

The findings of this thesis open several paths for ongoing study in the domain of legacy application modernisation:

7.4.1 Automated Migration Tooling

Research Opportunity: Extend the SRAPA (Spring RAP Automation) concept to **React code generation** from Eclipse RCP source code.

Specific Investigations:

- **Static Analysis for UI Pattern Recognition:** Design machine learning models trained on Eclipse RCP → React migration pairs to automatically infer:
 - SWT Composite layout structures → corresponding React component trees.
 - JFace data binding idioms → analogous React Hook Form usage.
 - Eclipse Command Framework creates → React context-driven command registries.
- **Incremental Migration Automation:** Build tooling that supports **stepwise migration** such that:
 - Static analysis detects "migration-ready" components exhibiting low coupling and high cohesion.
 - The tool emits React component skeletons annotated with TODO markers for manual enhancement.
 - A hybrid runtime setup enables progressive substitution of RCP views with web-based counterparts.

Expected Impact: Shorten the typical React migration duration for standard applications from 8–12 weeks to 4–6 weeks, thus reducing the effort required for adoption.

Technical Challenges:

- The dynamic plugin loading model of Eclipse RCP limits the effectiveness of static code analysis.
- SWT's imperative approach to widget construction is hard to translate into React's declarative JSX style.
- Making sure that the generated code complies with maintainability criteria (TDR <5%).

7.4.2 Polyglot Architecture Patterns for Hybrid Tool Suites

Research Opportunity: Formalise architectural patterns for **React + Wasm hybrid applications** where standard components use React and performance-critical components use Slint/WebAssembly.

Targeted Investigations:

- **Shared State Management:** Explore best-practice patterns for keeping state in sync between React components (JavaScript heap) and Wasm modules (linear memory):
 - SharedArrayBuffer for handling frequent updates.
 - Message-passing mechanisms for infrequent synchronisation.
 - Combined strategies that incorporate client-side caching.
- **Component Boundary Optimisation:** Develop metrics to establish optimal React/Wasm split:
 - Profiling tools to identify performance bottlenecks in React implementations.
 - Evaluate the costs and benefits of migrating selected components to Wasm.
 - Define criteria as to when the 80/20 split is appropriate (80% React, 20% Wasm).
- **Build Pipeline Integration:** Develop a unified toolchain that smoothly compiles both React (Webpack/Vite) and Wasm (wasm-pack) components within a single CI/CD pipeline.

Expected Impact: Allow organisations to implement a “progressive Wasm enhancement” approach—begin with a standard React application and, as performance demands grow, incrementally migrate critical hotspots to Wasm.

7.4.3 AI-Assisted Migration and Modernisation

Research Opportunity: Investigate **Large Language Model (LLM) applications** for Eclipse RCP modernisation tasks.

Specific Investigations:

Code Translation Assistance:

- Fine-tune LLMs (GPT-4, Claude, Llama) using a corpus of Eclipse RCP → React migration examples.
- Measure the accuracy of automatic SWT → JSX code conversion.
- Evaluate the fidelity of TypeScript type definitions automatically generated from Java classes.

Architectural Decision Support:

- Use migration case studies to develop models that can suggest suitable migration pathways.
- Automatically apply decision matrices driven by insights from codebase analysis.
- Create migration roadmaps that include estimates of required effort.

Documentation Generation:

- Auto-generate migration guides directly from legacy RCP source code.
- Build API documentation for the newly extracted headless services.
- Develop training content for developers moving to web-native technologies.

Technical Challenges:

- LLMs may fabricate APIs that do not exist (for example, by making up React components).
- Without human monitoring, generated code might fail to satisfy maintainability requirements.
- Ethical concern: Excessive dependence on AI-produced code can hinder developers' skill growth.

Expected Impact: Reduce manual migration effort by 30-40% for standard applications, allowing human engineers to focus on complex architectural decisions.

7.4.4 Accessibility and inclusive design in Migrated Applications

Research Opportunity: Investigate how migration from Eclipse RCP to web-native platforms impacts access to **users with disabilities**.

Specific Investigations:

- **ARIA Implementation Gaps:** Many SWT widgets in Eclipse RCP are missing appropriate ARIA attributes. Determine whether migrating to React/Wasm enhances or worsens compatibility with screen readers.
- **Keyboard Navigation Schemes:** Contrast Eclipse RCP's rich set of keyboard shortcuts with those available in web-based interfaces. Explore methods to maintain high efficiency for power users.
- **Colour Contrast and Theme Support:** Verify that web-based implementations continue to meet accessibility standards (WCAG 2.1 AA) for both light and dark themes.

Expected Impact: Protect against modernisation efforts inadvertently marginalising users with disabilities, which is especially critical for enterprise applications serving multiple user groups.

7.5 Concluding Remarks

The modernisation of Eclipse Rich Client Platform applications represents a representation of the broader challenge facing enterprise software engineering: how to preserve decades of accumulated intellectual property while adopting architectural paradigms fundamentally incompatible with legacy design assumptions. This thesis has contributed to the idea that successful migration requires going beyond simplistic dichotomies ("rewrite vs. reuse", "cloud vs. on-premises", "performance vs. maintainability") toward nuanced, empirically-grounded decision frameworks which acknowledge the multidimensional trade-off inherent in those transformations.

The central finding—that maximum code reuse may perpetuate rather than fix technical debt—challenges deeply-held assumptions in the software engineering community about the inherent value of preservation. The empirical evidence reveals that Eclipse RAP's 82% code reuse, while minimising initial development effort, carries forward architectural constraints (stateful server sessions, tight UI-logic coupling, monolithic state management) that fundamentally conflict with cloud-native principles of horizontal scalability, stateless design, and operational functionality. In contrast, React and Slint/WebAssembly—despite requiring 88-92% code rewrite—enable fundamental architectural modernisation that positions organisations to long-term sustainability in an increasingly cloud-centric software landscape.

The use-case-driven differentiation between React and Slint/WebAssembly provides a second critical insight: **performance requirements are not uniformly distributed** across application portfolios. The thesis validates that for >80% of Eclipse RCP tools (standard business applications with moderate UI complexity), React provides an optimal equilibrium of development velocity, ecosystem maturity, and acceptable performance. However, for the <20% of graphics-intensive applications that require sustained 60 FPS rendering and deterministic memory behaviour, Slint/WebAssembly's near-native performance characteristics justify the higher initial development investment and steeper learning curve. This finding enables organisations to adopt differentiated strategies rather than forcing all applications through a single migration pathway.

From a methodological perspective, the thesis shows the value of rigorous Design Science Research in addressing complex challenges in enterprise software engineering. By iterating through observational evaluation phases—universal maintainability assessment, architectural compatibility studies, and use-case-specific performance validation—the research produces prescriptive knowledge that is simultaneously generalisable (applicable beyond the automotive domain) and actionable (supplying clear decision matrices and implementation patterns). Integration patterns documented for VS Code and Eclipse Theia provide reusable architectural blueprints that reduce the "packaging problem" inherent in transitioning from OSGi bundles to web-based extension models.

For practitioners presently navigating Eclipse RCP modernisation choices, this thesis provides several directly applicable recommendations:

1. **Do not treat "lift-and-shift" as the final strategy:** Use Eclipse RAP only as an interim solution with a clearly defined end-of-life horizon (2–3 years), rather than as the long-term modernisation destination.
2. **Emphasise architectural decoupling over preserving existing code:** Focus resources on isolating headless business logic into platform-neutral services before committing to a specific migration route.
3. **Choose technologies based on concrete use cases:** Prefer React for typical business applications, and reserve Slint/WebAssembly for graphics-heavy tools that must consistently sustain 60 FPS.
4. **Build genuine web-native expertise:** Moving from Eclipse RCP to contemporary web architectures calls for substantial new skills that cannot be gained through short, superficial training sessions.

Looking toward the future, the swift advancement of web technologies (React Server Components, WebAssembly Component Model, WebGPU compute shaders) suggests that the performance margins documented in this thesis will continue to shift. Organisations must adopt continuous assessment practices, revisiting migration decisions as ecosystem maturity evolves. The decision framework presented here delivers a structured methodology for such re-evaluation, making certain that architectural choices continue aligned with both technological capacities as well as strategic business objectives.

In conclusion, the modernisation of Eclipse RCP applications is not just a technical task, but is more an organisational transformation. The empirical findings reported in this thesis offer a basis for strategic decision-making. For the automotive domain in particular, and for any industry facing comparable legacy modernisation pressures, this work outlines a roadmap for managing the shift from desktop-centric architectures to a web environment and a cloud-native landscape, embracing the architectural models that will underpin the next generation of software systems.

Bibliography

- [1] *The Future of the Eclipse Platform and Eclipse RCP*. Accessed: Dec. 13, 2025. EclipseSource. URL: <https://eclipsesource.com/blogs/2025/07/16/the-future-of-the-eclipse-platform-and-rcp/> (cit. on pp. 1–5, 17).
- [2] *Eclipse RCP Platform Benefits*. Accessed: Dec. 13, 2025. Eclipse Foundation. URL: <https://eclipseide.org/eclipse-platform/eclipse-platform-benefits/> (cit. on p. 1).
- [3] *Eclipse RCP based on the 3.x API - Tutorial*. Accessed: Dec. 13, 2025. Vogella GmbH. URL: <https://www.vogella.com/tutorials/Eclipse3RCP/article.html> (cit. on p. 2).
- [4] *SWT Evolve vs. VS Code*. Accessed: Dec. 13, 2025. Equo. URL: <https://www.equo.dev/blog/swt-evolve-vs-visual-studio-code> (cit. on pp. 2, 6, 10, 11).
- [5] *When should you build a web application vs. a thick client?* Accessed: Dec. 13, 2025. Stack Overflow. URL: <https://stackoverflow.com/questions/522422/when-should-you-build-a-web-application-vs-a-thick-client> (cit. on pp. 2–4).
- [6] *What is Technical Debt?* Accessed: Dec. 13, 2025. IBM. URL: <https://www.ibm.com/think/topics/technical-debt> (cit. on p. 3).
- [7] *Software Development Cost: A Guide to Legacy Systems, Rewrites, and Microservices*. Accessed: Dec. 13, 2025. Iterators. URL: <https://www.iteratorshq.com/blog/software-development-cost-a-guide-to-legacy-systems-rewrites-and-microservices/> (cit. on pp. 3, 4).
- [8] *Exploring the synergy between cloud computing and enterprise architecture: Challenges and opportunities*. Accessed: Dec. 13, 2025. International Journal of Science and Research Archive. URL: <https://ijsra.net/sites/default/files/IJSRA-2023-0416.pdf> (cit. on p. 4).
- [9] *How the Cloud Has Transformed Enterprise Architecture*. Accessed: Dec. 13, 2025. Workday. URL: <https://blog.workday.com/en-us/how-cloud-has-transformed-enterprise-architecture.html> (cit. on p. 4).
- [10] *What is Legacy Application Modernization?* Accessed: Dec. 13, 2025. IBM. URL: <https://www.ibm.com/think/topics/legacy-application-modernization> (cit. on pp. 4, 9).

- [11] *Cloud-Native or Lift-and-Shift?* Accessed: Dec. 13, 2025. AWS Enterprise Collection. URL: <https://medium.com/aws-enterprise-collection/cloud-native-or-lift-and-shift-99970053b25b> (cit. on p. 4).
- [12] *How to improve RCP applications startup time?* Accessed: Dec. 13, 2025. Eclipse Foundation. URL: <https://www.eclipse.org/forums/index.php/t/1083266/> (cit. on p. 5).
- [13] *Decoding Cloud-Native Architecture: Best Practices for Modernizing Legacy Applications.* Accessed: Dec. 13, 2025. ResearchGate. URL: https://www.researchgate.net/publication/393584036_Decoding_Cloud-Native_Architecture_Best_Practices_for_Modernizing_Legacy_Applications (cit. on p. 5).
- [14] *Migrating Eclipse and RCP Tools to the Web.* Accessed: Dec. 13, 2025. EclipseSource. URL: <https://eclipsesource.com/blogs/2025/07/30/migrating-eclipse-rcp-tools-to-web/> (cit. on pp. 6, 9, 11).
- [15] Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani. *Software Architecture: The Hard Parts.* Accessed: Dec. 22, 2025. URL: <https://www.oreilly.com/library/view/software-architecture-the/9781492086888/> (cit. on p. 6).
- [16] *Visual Studio Code Documentation.* Accessed: Mar. 12, 2026. Microsoft. URL: <https://code.visualstudio.com/docs> (cit. on p. 7).
- [17] *Visual Studio Code Extension API.* Accessed: Mar. 12, 2026. Microsoft. URL: <https://code.visualstudio.com/api> (cit. on p. 7).
- [18] *Visual Studio Code Marketplace.* Accessed: Mar. 12, 2026. Microsoft. URL: <https://marketplace.visualstudio.com/vscode> (cit. on p. 7).
- [19] *Visual Studio Code Extension API Limitations.* Accessed: Mar. 12, 2026. Microsoft. URL: <https://github.com/microsoft/vscode/wiki/Extension-API-Guidelines> (cit. on p. 7).
- [20] *VS Code Architecture Guide.* Accessed: Dec. 22, 2025. The Developer Space. URL: <https://thedeveloperspace.com/vs-code-architecture-guide/> (cit. on pp. 7, 20).
- [21] *Eclipse Theia: the Natural Cloud Migration Path for Eclipse Platform Adopters.* Accessed: Dec. 13, 2025. Eclipse Foundation. URL: <https://newsroom.eclipse.org/eclipse-newsletter/2022/may/eclipse-theia-natural-cloud-migration-path-eclipse-platform-adopters> (cit. on pp. 7, 24).
- [22] *Eclipse Theia - FAQ for Adopters and Users.* Accessed: Dec. 13, 2025. EclipseSource. URL: <https://eclipsesource.com/blogs/2019/12/24/eclipse-theia-ide-faq/> (cit. on pp. 7, 8, 11).
- [23] *Architecture Overview - Theia IDE.* Accessed: Dec. 13, 2025. Eclipse Foundation. URL: <https://theia-ide.org/docs/architecture/> (cit. on pp. 7, 11, 30).

- [24] *The Active Ecosystem of Eclipse Theia Adopters: A Tour of Diverse Tools and IDEs*. Accessed: Dec. 13, 2025. Eclipse Foundation. URL: <https://www.eclipse.org/topics/ide/articles/the-active-ecosystem-of-eclipse-theia-adopters> (cit. on p. 8).
- [25] *What is Replatforming?* Accessed: Dec. 13, 2025. Auth0. URL: <https://auth0.com/blog/what-is-replatforming/> (cit. on p. 9).
- [26] *Eclipse GLSPTM Graphical Language Server Platform for next-generation diagram editors*. Accessed: Dec. 13, 2025. Eclipse Foundation. URL: <https://eclipse.dev/glsp/> (cit. on pp. 9, 10, 12).
- [27] *Case study: Eclipse Rich Ajax Platform Use at CAS Software AG*. Accessed: Dec. 13, 2025. InfoQ. URL: <https://www.infoq.com/articles/eclipse-rap-casestudy/> (cit. on pp. 10, 11).
- [28] *Diagram Editors for Web-based Tools with Eclipse GLSP*. Accessed: Dec. 13, 2025. Eclipse Foundation. URL: <https://blogs.eclipse.org/post/brian-king/diagram-editors-web-based-tools-eclipse-glsp> (cit. on p. 10).
- [29] *How to reuse the Eclipse 4 IDE Perspective Features in an RCP Application*. Accessed: Dec. 13, 2025. Stack Overflow. URL: <https://stackoverflow.com/questions/16400757/how-to-reuse-the-eclipse-4-ide-perspective-features-in-an-rcp-application> (cit. on p. 10).
- [30] *TypeScript: JavaScript With Syntax For Types*. Accessed: Mar. 12, 2026. Microsoft. URL: <https://www.typescriptlang.org/docs/> (cit. on p. 11).
- [31] *React – A JavaScript library for building user interfaces*. Accessed: Mar. 12, 2026. Meta (Facebook). URL: <https://reactjs.org/docs/getting-started.html> (cit. on p. 11).
- [32] *PhosphorJS: A TypeScript library for building interactive multi-panel user interfaces*. Accessed: Mar. 12, 2026. PhosphorJS Contributors. URL: <https://github.com/phosphorjs/phosphor> (cit. on p. 11).
- [33] *Webpack – Module Bundler*. Accessed: Mar. 12, 2026. Webpack Contributors. URL: <https://webpack.js.org/concepts/> (cit. on p. 11).
- [34] *Modern Web Development: Understanding the Ecosystem*. Accessed: Mar. 12, 2026. Smashing Magazine. URL: <https://www.smashingmagazine.com/2020/05/modern-web-development/> (cit. on p. 11).
- [35] *A Vision for Flexible GLSP-based Web Modeling Tools*. Accessed: Dec. 13, 2025. arXiv. URL: <https://arxiv.org/pdf/2307.01352> (cit. on p. 12).
- [36] *Eclipse Theia is the next generation Eclipse RCP?!* Accessed: Dec. 13, 2025. EclipseSource. URL: <https://eclipsesource.com/blogs/2022/03/16/eclipse-theia-is-the-next-generation-eclipse-rcp/> (cit. on p. 18).
- [37] *Eclipse Remote Application Platform*. Accessed: Nov. 8, 2025. Eclipse Foundation. URL: <https://www.eclipse.org/rap> (cit. on p. 18).

- [38] *Choosing a RAP Application Setup*. Accessed: Nov. 8, 2025. Eclipse Foundation. URL: <https://help.eclipse.org/latest/topic/org.eclipse.rap.doc/guide/articles/application-setup.html> (cit. on p. 19).
- [39] *The RAP Client*. Accessed: Nov. 8, 2025. Eclipse Foundation. URL: <https://download.eclipse.org/rt/rap/doc/2.1/guide/articles/client.html> (cit. on p. 20).
- [40] *RAP 2.0 - New and Noteworthy*. Accessed: Nov. 8, 2025. Eclipse Foundation. 2013. URL: <https://eclipse.dev/rap/noteworthy/2.0/> (cit. on p. 20).
- [41] *Extension Host / Visual Studio Code Extension API*. Accessed: Nov. 8, 2025. Microsoft. URL: <https://code.visualstudio.com/api/advanced-topics/extension-host> (cit. on p. 21).
- [42] *Native-Like Cross-Platform Desktop Application Development with Electron*. Accessed: Nov. 8, 2025. Appilian. URL: <https://appilian.com/native-like-cross-platform-desktop-application-development-with-electron/> (cit. on p. 21).
- [43] *Supporting Remote Development*. Accessed: Dec. 22, 2025. Microsoft. URL: <https://code.visualstudio.com/api/advanced-topics/remote-extensions> (cit. on p. 22).
- [44] *Webviews / Visual Studio Code Extension API*. Accessed: Nov. 8, 2025. Microsoft. 2025. URL: <https://code.visualstudio.com/api/ux-guidelines/webviews> (cit. on p. 23).
- [45] *JSON-RPC*. Accessed: Nov. 8, 2025. JSON-RPC. URL: <https://json-rpc.org/> (cit. on p. 23).
- [46] *Eclipse Theia vs. Code OSS*. Accessed: Nov. 8, 2025. EclipseSource. Sept. 2023. URL: <https://eclipsesource.com/blogs/2023/09/08/eclipse-theia-vs-code-oss/> (cit. on p. 25).
- [47] *Authoring Theia Extensions*. Accessed: Dec. 22, 2025. Eclipse Foundation. URL: <https://theia-ide.org/docs/extensions/> (cit. on p. 25).
- [48] *SWT Evolve: Modernize your Eclipse RCP applications*. Accessed: Dec. 12, 2025. Equo. URL: <https://www.equo.dev/products/swt-evolve> (cit. on p. 25).
- [49] *Vaadin Framework*. Accessed: Mar. 12, 2026. Vaadin Ltd. URL: <https://vaadin.com/docs/latest/> (cit. on p. 27).
- [50] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. Accessed: Dec. 12, 2025. San Francisco, CA: No Starch Press, 2019. ISBN: 978-1593278281 (cit. on p. 29).
- [51] Amit Levy et al. “Ownership is Theft: Experiences Building an Embedded OS in Rust”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Accessed: Dec. 12, 2025. ACM, 2017, pp. 21–36. DOI: 10.1145/3062341.3062349 (cit. on p. 29).

- [52] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. “Bringing the Web up to Speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Accessed: Dec. 12, 2025. ACM, 2017, pp. 185–200. DOI: 10.1145/3062341.3062363 (cit. on p. 29).
- [53] Slint Technical Team. *Slint: Design Goals and Performance Benchmarks for Embedded UI*. Accessed: Dec. 13, 2025. Slint.dev. 2025. URL: <https://docs.slint.dev/latest/docs/slint/guide/platforms/embedded/> (cit. on p. 30).
- [54] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. “Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code”. In: *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Accessed: Dec. 13, 2025. Renton, WA: USENIX Association, 2019, pp. 107–122. URL: <https://www.usenix.org/conference/atc19/presentation/jangda> (cit. on p. 30).