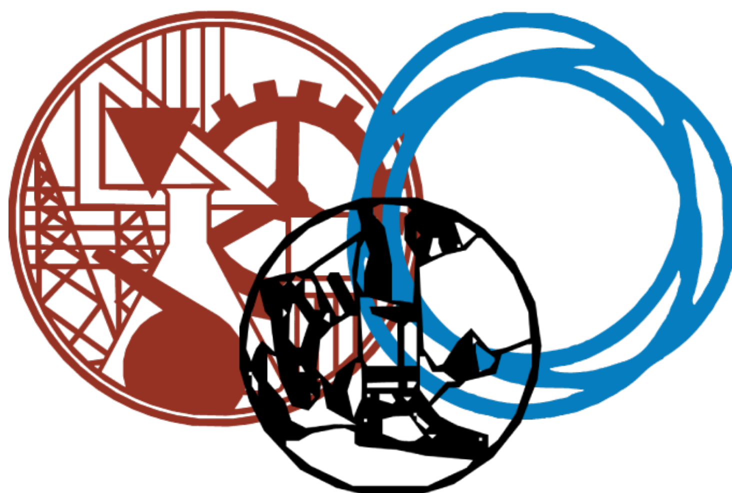
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Department of Electronical Engineering, Telecommunications and Computerss



Engineering the *SHIDA* Super-app

Research, Design and Development of a Literature-centered Social Network with E-commerce and E-learning

DIOGO FILIPE RICARDO RIBEIRO
(graduate in Informatics and Multimedia Engineering)

Dissertation of scientific nature to obtain a Master's degree
in Informatics and Multimedia Engineering

Supervisor:
Prof. Pedro Viçoso Fazenda, PhD

Co-Supervisor:
Eng. Theodros Mehari, CEO of Rhoda Engineering AB

Jury:
Prof. Rui Manuel Feliciano de Jesus, PhD (Chairperson)
Prof. Carlos Jorge Sousa Gonçalves, PhD
Prof. Pedro Viçoso Fazenda, PhD

December 2023

Acknowledgment

I would like to express my sincere gratitude to the individuals who played a significant role in completing this dissertation. Their guidance, teachings, and support were instrumental in shaping the approach and direction of this work.

I extend my heartfelt appreciation to my advisor, Engineer Pedro Fazenda, for his valuable teachings on mobile app development, which greatly influenced my understanding of the mobile platform. I am grateful for his continuous support and providing me with the freedom to explore and excel during my journey.

I am also deeply thankful to Engineer Rui Jesus, the Master's program coordinator, for his teachings on front-end frameworks, hybrid app development, and user experience design. His knowledge and insights have been invaluable in shaping the framework and approach employed in this dissertation.

I would also like to acknowledge the CEO of Rhoda Engineering, Engineer Theodros Mehari, for his generous support and resources during my time in Sweden. His guidance and encouragement were crucial in making my stay enjoyable and conducive to personal and professional growth. It was through his visionary ideas, extensive market study, and valuable connections that this project came to fruition.

Furthermore, I express my gratitude to the European Commission and the Erasmus+ program for providing me with this remarkable opportunity and the financial support that made it possible. This experience broadened my horizons and allowed me to engage with professionals from diverse cultures and backgrounds.

Lastly, I thank my family, friends, and colleagues for their unwavering support, encouragement, and understanding throughout this journey. Their presence and belief in my abilities have been a constant source of inspiration.

I am immensely grateful to everyone who contributed to my academic and professional development. Your teachings, support, and encouragement were vital in shaping the outcome of this dissertation.

ABSTRACT

In the realm of software engineering and web development, the *SHIDA* superapp serves as a case study navigating complex challenges. This research explores software architecture, agile methodologies, and the evolving landscape of full-stack development. It scrutinizes the journey from requirement analysis to testing while addressing issues like technological debt and scope creep. By examining the *SHIDA* project, this study uncovers obstacles and opportunities, offering insights into code maintainability, scalability, and performance optimization in modern software development.

Keywords: *Software Engineering, Software Architecture, Web Development, Full-Stack Development, Agile Methodologies, UI/UX Design, System Design, User-Centric Design, Version Control, Monolithic Repository, CSS, Application Frameworks, Microservices, Requirement Analysis, Test-Driven Development, Cloud Infrastructure, DevOps, CI/CD, Emerging Technologies, Code Maintainability, Scalability, Cross-Platform Development, Development Tools, SDLC, Code Refactoring, API Design, Serverless Architecture*

// Resumo (abstract translated to Portuguese)

No domínio da engenharia de software e desenvolvimento Web, a "super-aplicação" *SHIDA* serve como exemplo para a abordagem a desafios complexos. Este documento explora arquitetura de *software*, metodologias ágeis e a evolução do desenvolvimento *full-stack*. Ao longo deste, é examinado o processo desde a análise de requisitos até os testes, abordando questões como a dívida tecnológica e o excesso de âmbito. Ao examinar o projeto *SHIDA*, este estudo revela tanto obstáculos como oportunidades, fornecendo critérios para melhor manutenção de código, escalabilidade e otimização de desempenho no desenvolvimento de *software* moderno.

Palavras-chave: *Engenharia de Software, Arquitetura de Software, Desenvolvimento Web, Desenvolvimento Full-Stack, Metodologias Ágeis, Design de Interfaces e Experiência do Utilizador, Design de Sistemas, Design Centrado no Utilizador, Controlo de Versão, Repositório Monolítico, CSS, Frameworks Aplicacionais, Microserviços, Análise de Requisitos, Desenvolvimento Orientado a Testes, Infraestrutura na Nuvem, DevOps, CI/CD, Tecnologias Emergentes, Manutenção de Código, Escalabilidade, Desenvolvimento Multiplataforma, Ferramentas de Desenvolvimento, Ciclo de Vida de Desenvolvimento de Software, Refatorização de Código, Design de Interfaces Aplicacionais, Arquitetura Serverless*

Table of contents

I Introduction **1**
I.1 Overview 2
I.1.1 Motivation 2
I.1.2 Problem Statement 3
I.1.3 Objectives 4
I.1.4 Research Questions 5
I.1.5 Contributions 6
I.1.6 Document Structure 7
I.2 Context 9
I.2.1 Company 9
I.2.2 Target Audience 11
I.2.3 Product Structure 12
I.2.4 Actors 14

II State-of-the-Art **15**
II.1 Programming Languages 16
II.2 Technology Stacks 17
II.2.1 “Frontend” and “Backend” 18
II.2.2 History of Tech Stacks 18
II.3 The Web Browser 19
II.3.1 Progressive Web Apps 20
II.3.2 Performance Metrics 21
II.3.3 Different “Flavors” 21
II.4 Application Layer 22
II.4.1 Distribution Patterns 23
II.4.2 Structural Patterns 24
II.4.3 Communication Patterns 25
II.4.4 Communication Protocols 26
II.4.5 API-related Tools 27
II.5 Presentation Layer 28
II.5.1 Framework Architectures 28
II.5.2 Application Structure 29
II.5.3 Rendering Strategies 29
II.5.4 Data Loading 31
II.6 The JavaScript Ecosystem 33
II.6.1 Frameworks 33
II.7 Other Languages on the Web 35

II.8 Mobile Platforms 35
II.8.1 Navigation 36
II.8.2 User Actions 37
II.8.3 Hybrid Frameworks 37
II.9 Desktop Platforms 38
II.10 Data Layer 39
II.11 Networking 40
II.12 Security 40
II.13 Server Environment, Deployment, and Hosting 41
II.14 Software Factory 42
II.15 User Interactivity Design 45
II.15.1 Evolution of CSS 45
II.15.2 UI Libraries 46
II.15.3 UI Trends 47
II.16 Software Development Strategies 48
II.16.1 Organization of code 49
II.16.2 Project Management Tools 50
II.16.3 Software Diagrams 50
II.16.4 Deployment Strategies 50
II.17 E-commerce 51
II.18 Social Networking 52
II.19 Recommender Systems 52
II.20 Book Reader Software 53
II.21 Audiobook Player Software 53
II.22 Similar Products 54
II.22.1 What is a super-app? 54
II.22.2 WeChat (General) 55
II.22.3 Kindle & Audible (Digital Library) 55
II.22.4 Clubhouse, Twitter Spaces & Discord (Talk Rooms) 56
II.22.5 Meetup (Matchmaking) 57
II.22.6 Medium (Knowledge Hub) 57
II.22.7 Masterclass (Knowledge Hub) 57
II.22.8 Goodreads (Digital Library) 58
II.22.9 Tinder/Bumble (Matchmaking) 58
II.22.10 Novellic (Book Clubs) 58
II.22.11 itch.io (Graphics/Games) 58

III Methodology 59
III.1 Project Requirements 60
III.1.1 General, High-Priority Requirements 60

III.1.2 Per Service 62
III.2 Software Development Process 66
III.2.1 Scaled Agile Framework (SAFe) 66
III.2.2 Large-Scale Scrum: More with LeSS 67
III.2.3 DevSecOps and CI/CD 68
III.3 Challenges 69
III.3.1 Over-Engineering 69
III.3.2 Chesterton's Fence 70
III.3.3 Analysis Paralysis 71
III.3.4 Scope Creep 72
III.3.5 Complexity Budget 72
III.3.6 Technological Debt 73
III.3.7 Future Thinking 74
III.3.8 Premature Optimizations 74
III.4 Comparative Analysis of Application Frameworks 75
III.4.1 Framework Evaluation Criteria 75
III.5 Technology Selection 76
III.5.1 Diagram Tools 77
III.5.2 Setting up Jira and Confluence 77
III.5.3 Software Anatomy Diagrams 77
III.5.4 Code Organization 78
III.5.5 Test framework 80
III.5.6 Human Resources Organization 80
III.5.7 The definition of done 82
III.5.8 Coding guidelines 82
III.6 Architectural Choices 82
III.6.1 Comparing Clouds 83
III.6.2 Data Access Layer 84
III.6.3 Business Layer 85
III.6.4 Presentation Layer 87

IV Development 89
IV.1 Preliminary Phase 90
IV.2 API Development 90
IV.2.1 Iteration 1: Express.js and Docker Compose 90
IV.2.2 Iteration 2: NestJS 90
IV.2.3 Iteration 3: Next.js API Serverless Functions 91
IV.2.4 Final Iteration: Django and Medusa 91
IV.3 Web App 92
IV.3.1 Iteration 1: React Native for Web 92

IV.3.2 Iteration 2: Nuxt ... 94
IV.3.3 Setting up PWA ... 95
IV.3.4 Setting up Internationalization ... 96
IV.3.5 Microfrontends Exploration ... 97
IV.3.6 Iteration 3: Flutter ... 97
IV.3.7 Final Iteration: Next.js ... 97
IV.4 Setting up AWS EC2 and GitLab ... 103
IV.5 Setting up VPC ... 104
IV.6 Setting up VPN and Certificates ... 105
IV.7 Setting up the Monorepo ... 106
IV.8 Developer Communication ... 107

V Conclusions and Future Work ... 108
V.1 Summary of Findings ... 109
V.2 Future Directions ... 111
V.3 Conclusion ... 113

References ... 115

A Entity-relationship diagram ... 117

B Functional and non-functional architecture diagram ... 118

C Meta-structure for the SHIDA monorepo ... 119

D Excalidraw whiteboard used to plan Jira tasks ... 120

List of figures

Figure 1: Starting point for the *SHIDA* website 10
Figure 2: Context diagram for the *SHIDA* superapp. 14
Figure 3: Flowchart depicting the core processes within the WebKit engine (source: Garsiel and Irish (2011)) 22
Figure 4: Sequence diagrams for different models for rendering a website .. 30
Figure 5: Landing page for presentation of the iPhone 14, displaying prominent use of Bento grids (Source: https://www.apple.com/iphone/) 48
Figure 6: “Essential” configuration of SAFe 5 (Source: https://v5.scaledagileframework.com/) 67
Figure 7: DevSecOps pipeline (Source: https://www.qentelli.com/thought-leadership/insights/devsecops-pipeline-factors) 68
Figure 8: Suggested profiles for hiring 81
Figure 9: First iteration of the Anatomy Diagram 93
Figure 10: Second iteration of the Anatomy Diagram 95
Figure 11: The *SHIDA* prototype, installed as a Progressive Web App on Android (accessible at https://standalone-eta.vercel.app/) 96
Figure 12: Third iteration of the Anatomy Diagram 99
Figure 13: The *SHIDA* prototypes, implemented in Figma 100
Figure 14: Diagram of the user’s planned navigation through the website 101
Figure 15: Book card grid: more complicated than it seems 102
Figure 16: Flowchart explaining client-side dynamic input validation 103
Figure 17: Informal network diagram of the *SHIDA* project 105
Figure 18: Excalidraw whiteboard used to keep track of system design . . 107

List of acronyms

ACID – Atomicity, Consistency, Isolation, and Durability. 39

ACL – Access Control List. 104

AJAX – Asynchronous JavaScript and XML. 32

AMQP – Advanced Message Queuing Protocol. 27

API – Application Programming Interface. 18, 19, 20, 22, 23, 25, 27, 28, 31, 32, 34, 35, 43, 86, 90, 91, 96, 102

AWS – Amazon Web Services. 39, 40, 42, 83, 84, 98, 104

CA – Certificate Authority. 40, 106

CD – Continuous Delivery: Also called Continuous Deployment 18, 49, 60, 68, 78, 79, 92, 103

CDN – Content Delivery Network. 24, 41, 60

CI – Continuous Integration. 18, 49, 60, 68, 78, 79, 92, 103

CLI – Command-Line Interface. 46, 106

CMS – Content Management System. 42, 52, 86, 112

CORS – Cross-origin Resource Sharing. 41

CQRS – Command Query Responsibility Segregation. 25

CRM – Customer Relationship Management. 51

CRUD – Create, Read, Update, Delete. 26

CSR – Client-Side Rendering. 29, 30, 31

CSS – Cascading Style Sheets. 45, 46, 47, 99

CVE – Common Vulnerabilities and Exposures. 41

DBMS – Database Management System. 39

DDoS – Distributed Denial-of-Service. 41

DMZ – Demilitarized Zone. 41, 104

DNS – Domain Name System. 40, 106

DRM – Digital Rights Management. 44, 53

DRY – Don't Repeat Yourself. 46

DX – Developer Experience. 92

FCP – First Contentful Paint. 21

FOSS – Free and Open Source Software. 106

GCP – Google Cloud Platform. 42, 83

GDPR – General Data Protection Regulation. 44, 61

GUI – Graphical User Interface. 39

HLS – HTTP Live Streaming. 40

HTML – Hypertext Markup Language. 19, 28, 29, 30, 31, 45, 46, 47, 95

HTTP – Hypertext Transfer Protocol. 26, 27, 35, 40

HTTPS – Hypertext Transfer Protocol Secure. 29, 40

IDE – Integrated Development Environment. 36

IMAP – Internet Message Access Protocol. 43

IP – Internet Protocol. 40, 104

ISR – Incremental Static Regeneration. 31

IaC – Infrastructure-as-Code. 41, 105

IoT – Internet of Things. 25, 27

JSON – JavaScript Object Notation. 20, 32, 39

JVM – Java Virtual Machine. 16

LCP – Last Contentful Paint. 21

LeSS – Large-Scale Scrum. 60, 67

MPA – Multi-Page Application. 29

MVC – Model-View-Controller. 28

MVP – Model-View-Presenter. 28

MVVM – Model-View-ViewModel. 28

OAS – OpenAPI Specification. 27

ORM – Object-Relational Mapping. 39, 40, 85

OS – Operating System. 35, 36, 38, 39

OSI – Open Systems Interconnection. 40

OWASP – Open Worldwide Application Security Project. 41

P2P – Peer-to-peer. 24

PI – Product Increment. 67, 78

PWA – Progressive Web App. 19, 20, 21, 92, 95

RBAC – Role-Based Access Control. 43, 61

RDBMS – Relational Database Management System. 39

REST – Representational State Transfer. 19, 26, 27, 36

RPC – Remote Procedure Call. 16, 26

RTP – Real-time Transport Protocol. 40

RoR – Ruby on Rails. 18

SAFe – Scaled Agile Framework: for Lean Enterprises 60, 66, 67

SDK – Software Development Kit. 36

SDLC – Software Development Lifecycle. 42

SEO – Search Engine Optimization. 21, 29, 31

SMTP – Simple Mail Transfer Protocol. 43

SOA – Service-Oriented Architecture. 24

SOAP – Simple Object Access Protocol. 26

SPA – Single-Page Application. 29, 30

SQL – Structured Query Language. 28, 39

SSE – Server-Sent Events. 30, 32

SSG – Static Site Generation. 30, 31, 34, 43

SSH – Secure Shell. 104

SSL – Secure Sockets Layer. 40, 106

SSR – Server-Side Rendering. 30, 31

TDD – Test-Driven Development. 49

TLS – Transport Layer Security. 40, 106

UI – User Interface. 28, 29, 34, 36, 45, 46, 47, 98, 100

UML – Unified Modeling Language. 48, 50, 111

URL – Uniform Resource Locator. 37

UX – User Experience. 45, 61, 100

VCS – Version Control Systems. 49

VIPER – View-Interactor-Presenter-Entity-Router. 28

VPN – Virtual Private Network. 105, 106

VPS – Virtual Private Server. 41

WAF – Web Application Firewall: OSI Layer 7 41, 61

WWW – World Wide Web. 19

XML – Extensible Markup Language. 26, 39

XSS – Cross-site Scripting. 41

I Introduction

This chapter serves as an introduction to the thesis, providing an overview of the research's context, motivation, objectives, and contributions. It sets the stage for the subsequent chapters, offering a clear roadmap of the study's goals and significance.

I.1 Overview

In this section, we delve into the motivation behind the research, exploring the rapid advancements in technology and the growing complexity of software engineering and web development in our digital age. We examine the challenges and opportunities that developers face in creating large-scale web applications, with a focus on the *SHIDA* superapp as a case study. This chapter aims to provide readers with a comprehensive understanding of the research's scope and objectives, offering a glimpse into the multifaceted world of modern software development.

I.1.1 Motivation

In an era defined by rapid technological advancements and an ever-expanding digital landscape, software engineering and web development have become pivotal domains. The demand for innovative and efficient solutions reached unprecedented heights during the COVID-19 pandemic, as businesses and individuals increasingly relied on digital platforms for communication, commerce, and information. Even in the aftermath of the pandemic, the trend continues. According to recent statistics from Statista (sourced from Dixon (2023)), the demand for social networking hasn't yet had a year of decline. This global survey reveals that the daily time spent on social media by internet users worldwide increased from an average of 147 minutes in 2022 to 151 minutes in 2023.

During development of the *SHIDA* project, the focal point of our research, we came to realize that software development, though promising in its transformative potential, is often plagued by a myriad of challenges. These challenges range from architectural decisions and development methodologies to the intricate details of user-centric design. As developers, we grapple not only with the complexities inherent in modern full-stack development but also with issues like technological debt and scope creep, which can hinder progress and compromise the quality of our creations.

Therefore, this study seeks to address these challenges head-on. It aims to dissect the multifaceted aspects of software engineering, using the *SHIDA* project as a living example. Through a comprehensive examination of this project, we intend to uncover both the obstacles encountered and the opportunities harnessed. By doing so, we aspire to contribute valuable insights that can guide developers and organizations towards effective code maintainability, scalability, and performance optimization in the dynamic landscape of modern software development.

In the following sections, we lay out the vision for the *SHIDA* project, delving into software architecture, agile methodologies, and the ever-evolving world of web development. Our objective is to provide a holistic understanding of the challenges and innovations that shape this domain, ultimately fostering our collective knowledge and capabilities in developing sophisticated and feature-rich applications from the ground up.

I.1.2 Problem Statement

In the realm of software engineering and web development, the *SHIDA* project unfolds as a comprehensive case study that navigates through multifaceted challenges. This study explores software architecture, encompassing monolithic repositories and microservices, alongside the effects of agile methodologies. It delves into the evolving landscape of full-stack development, highlighting the significance of user-centric design and addressing emerging technologies and frameworks. Throughout this thesis, we dissect the journey from requirement analysis to implementation and testing, examining and explaining the decisions made along the way.

The *SHIDA* project is a pioneering initiative that aspires to create a social network seamlessly intertwined with e-commerce, e-learning, and literature. This intricate fusion poses unique challenges as it endeavors to offer:

- **E-commerce Integration:** Developing a robust e-commerce system that facilitates transactions for literature-related products, taking into account payment methods commonly used in the region.
- **Office Tools for Offline Document Reading:** Offering users the capability to access and interact with literary content even when offline, incorporating office tools for document reading.
- **E-Learning Platform:** Providing a dynamic e-learning environment that enables users to access educational content and resources seamlessly.
- **Social Networking Features:** Implementing social networking components to foster community engagement and interaction among users.
- **User-Centric Design:** Prioritizing user experience (UX) design to provide an intuitive and inclusive platform, especially for users accessing the internet via mobile devices.
- **Cross-Platform Development:** Implementing a strategy for delivering the application across various platforms, considering the predominance of mobile devices for internet access in the region.

- **Scalability:** Ensuring the platform's scalability to accommodate a potentially large user base, balancing performance and resource optimization.
- **Content Localization:** Adapting content and user interfaces to multiple languages and cultures, enhancing accessibility and engagement.
- **Technological Infrastructure:** Leveraging the latest technologies and frameworks while addressing the infrastructure limitations and connectivity challenges often encountered in the African context.
- **Local Partnerships:** Establishing partnerships with local publishers, authors, and educational institutions to enrich the platform's content and services.
- **Regulatory Compliance:** Navigating the regulatory landscape and data privacy requirements specific to the African countries where *SHIDA* intends to operate.

This study dissects how the *SHIDA* project meticulously addresses the intricate blend of literature, e-commerce, e-learning, office tools, and social networking, while also exploring how it confronts the challenges presented by this harmonious amalgamation. The thesis uncovers not only the obstacles encountered but also the opportunities harnessed, forging a path towards effective code maintainability, scalability, and performance optimization in the modern age of software development.

I.1.3 Objectives

The primary objectives of this thesis are to:

1. **Analyze Software Architecture:** Examine the software architecture choices made in the *SHIDA* project, including the effect of adopting monolithic repositories and microservices. Evaluate their effectiveness in addressing the unique requirements of the social network with integrated e-commerce, e-learning, and literature.
2. **Explore Agile Methodologies:** Investigate the application of agile methodologies in the development process of the *SHIDA* project. Assess how agile practices contribute to project adaptability and the ability to respond to evolving user needs.
3. **Address Technological Challenges:** Examine the strategies employed to address technological challenges, such as infrastructure limitations, connectivity issues, and cross-platform compatibility, while leveraging the latest technologies and frameworks.

4. **Establish User-Centric Design:** Establish a user-centric design approach for the SHIDA project by thoroughly understanding the target user personas, their characteristics, goals, and expectations, ensuring an intuitive and inclusive platform, particularly for users accessing the internet via mobile devices.
5. **Explore Content Localization:** Investigate strategies for content and user interface localization tailored to the specific needs and challenges of the *SHIDA* project's less common target audience, including those in regions with limited internet access, preference for local languages, and varying technological literacy levels.
6. **Navigate Regulatory Compliance:** Investigate how the *SHIDA* project navigates the complex regulatory landscape and data privacy requirements specific to the African countries where it operates.

By pursuing these objectives, this thesis aims to provide valuable insights into the multifaceted challenges faced by software development projects in the context of creating comprehensive, large-scale, and multi-faceted web applications, with a focus on the technology selection, research and development, and strategic decision-making involved in the development process.

I.1.4 Research Questions

In alignment with the defined objectives, this thesis seeks to address specific research questions that provide a structured approach to investigating the challenges and strategies within the *SHIDA* project:

Software Architecture (RQ1)

What are the key architectural choices made in the *SHIDA* project, and how do these choices impact the project's ability to meet its unique requirements? How does the project's software architecture facilitate the integration of social networking, e-commerce, e-learning, and literature?

Agile Methodologies (RQ2)

How are agile methodologies, such as sprint planning and continuous integration, applied within the development process of the *SHIDA* project? How do these agile practices contribute to the project's adaptability and responsiveness to evolving user needs?

Technological Challenges (RQ3)

What strategies are employed in the *SHIDA* project to address technological challenges related to infrastructure limitations, connectivity issues, and cross-platform compatibility, while leveraging the latest technologies and frameworks?

User-Centric Design (RQ4)

To what extent is user-centric design implemented in the *SHIDA* project to ensure an intuitive and inclusive platform, particularly for users accessing the internet via mobile devices in regions with diverse linguistic and cultural backgrounds?

Content Localization (RQ5)

How does the *SHIDA* project adapt its content and user interfaces to accommodate its unique context, enhancing accessibility and engagement for users across diverse languages and cultures?

Regulatory Compliance (RQ6)

How does the *SHIDA* project navigate the complex regulatory landscape and data privacy requirements specific to the African countries where it operates, with a focus on privacy laws, content regulations, and data protection measures?

These research questions guide the exploration of critical aspects within the *SHIDA* project, facilitating a comprehensive analysis of the challenges and opportunities encountered during its development. Through the investigation of these questions, this thesis aims to contribute valuable insights to the field of software engineering and web application development, particularly concerning the development of large-scale, multifaceted web applications.

I.1.5 Contributions

This thesis attempts to make several contributions to the field of software engineering, web application development, and the understanding of complex projects like the *SHIDA* initiative. The key contributions are as follows:

1. **Analysis of Software Architecture:** This research provides a detailed analysis of the *SHIDA* project's software architecture choices, specifically examining the use of monolithic repositories and microservices. It sheds light on the impact of these architectural decisions on the project's ability to

meet its unique requirements, offering valuable insights for future projects facing similar challenges.

2. **Exploration of Agile Methodologies:** By investigating the application of agile methodologies within the *SHIDA* project, this thesis contributes to a deeper understanding of how agile practices can enhance adaptability and responsiveness in the development of large-scale, multifaceted web applications. These insights can benefit both practitioners and researchers in the field.
3. **Evaluation of User-Centric Design:** The evaluation of user-centric design principles in the *SHIDA* project contributes to a better understanding of how to create intuitive and inclusive platforms, particularly for users accessing the internet via mobile devices in regions with diverse cultural and socioeconomic backgrounds.
4. **Addressing Technological Challenges:** This thesis explores strategies employed to address technological challenges in the context of infrastructure limitations, connectivity issues, and cross-platform compatibility. The findings provide guidance for developers and organizations working on projects in regions with similar challenges.
5. **Optimizing Content Localization:** The examination of how the *SHIDA* project adapts content and user interfaces for offline-readiness and accessibility in regions with weak internet infrastructure contributes to a better understanding of content localization strategies in diverse contexts.
6. **Navigating Regulatory Compliance:** By investigating how the *SHIDA* project navigates complex regulatory landscapes and data privacy requirements in African countries, this thesis contributes insights into compliance strategies and data protection measures for projects operating in diverse legal environments.

These contributions collectively advance the understanding of the challenges and opportunities inherent in developing comprehensive, large-scale web applications that integrate social networking, e-commerce, e-learning, literature, and address the unique contexts of regions with varying infrastructural, linguistic, and regulatory landscapes.

I.1.6 Document Structure

This document is structured into several chapters, each serving a distinct purpose in presenting the research and its outcomes.

Chapter 1: Introduction

In the current chapter, the research is introduced, encompassing aspects like motivation, problem statement, objectives, research questions, contributions, and the document's structure. Additionally, the research's contextual background, including details about the company, target audience, and project structure, is explored.

Chapter 2: State-of-the-art

This chapter provides an overview of the current state of technology stacks, programming languages, web browsers, and web frameworks. It covers components of technology stacks, distinctions between backend and frontend technologies, primary programming languages, and noteworthy mentions. The section on web browsers discusses their high-level structure, rendering processes, notable features and web APIs, performance considerations, and the current state of web browsers. Furthermore, it explores the JavaScript ecosystem and related web frameworks and technologies.

Chapter 3: Methodology

Chapter 3 delves into the research methodology used in the study. It outlines the methods employed for data collection and analysis, providing insights into the research process.

Chapter 4: Development

This chapter discusses the development process, highlighting key findings related to technology stacks, programming languages, web browsers, and web frameworks. It presents insights and discoveries drawn from the research data, allowing for a comprehensive understanding of the development aspects.

Chapter 5: Conclusion

Chapter 5 serves as the conclusion of the document. It includes a summary of research findings, implications, and recommendations for future work. The conclusion section summarizes the main takeaways and emphasizes the contributions of the research.

References

The document concludes with a reference section that lists all the sources and references cited throughout the document, providing readers with a comprehensive list of information sources.

I.2 Context

The context section of a document provides an understanding of the purpose of the project, its actors and stakeholders and how it fits into the broader context of the thesis's goals and objectives.

I.2.1 Company

Rhoda Engineering AB, headquartered in Gothenburg, Sweden, specializes in Distributed and Centralised Embedded System development and offers a range of services. These services encompass software development, system and software architecture, data analysis, DevOpt services, and software testing.

Rhoda Engineering's expertise mainly extends to the automotive industry as the industry has become the melting point of integrated technologies, namely, Advanced Driver Assist System (ADAS) and Autonomous Drive System (AD), Supplementary Restraint System (SRS), infotainment systems, Google Automotive Service (GAS), and propulsion systems. Their team of engineers partners with clients to design and develop efficient solutions, excelling in areas such as Software engineering, System Design, requirement management, change management, and software release and configuration management.

In addition to consultancy, Rhoda Engineering offers educational and training programs. These courses cover various topics, including automotive electronics topology, data communication, AUTOSAR basics, test method development, and test analysis. These programs aim to deepen understanding in agile software development and development team management.

Rhoda Engineering has a notable track record of collaboration with prominent companies in the industry, including Volvo Cars, Volvo Group, Nvidia, ECARX, Ericsson, Zenseact, and Bosch. They also maintain fruitful partnerships, notably with the Chalmers University of Technology.

This thesis project aligns with Rhoda Engineering's commitment to providing comprehensive software solutions, addressing the company's growing interest in frontend development and web technologies.

SHIDA Networks operates as a subsidiary of Rhoda Engineering, functioning like a startup. It currently includes a publishing branch responsible for translating numerous Eritrean books into English, aimed at expanding their reach to a global audience. Additionally, there is a WordPress-based static news

website, *SHIDA Media*, which focuses on Eritrean content. Figure 1 shows the aforementioned website, which is a starting point for the *SHIDA* project.

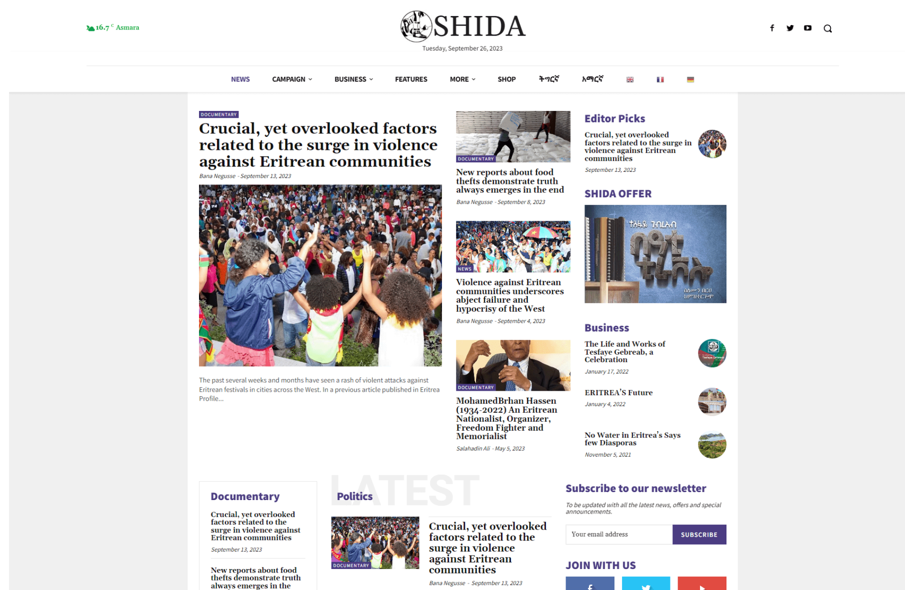


Figure 1: Starting point for the *SHIDA* website

Role of Eng. Mehari

Eng. Mehari, as the CEO of Rhoda, played a pivotal role in this project, leveraging extensive expertise in Embedded Systems and the automotive industry, with a particular focus on collaborations with Volvo Cars. However, as could be expected, his familiarity with Web Development and the specific technologies used in this project was limited.

In his higher-level authority position, Eng. Mehari shouldered a range of critical responsibilities. These encompassed establishing project requirements, shaping the software development strategy, defining the business model, and providing a clear vision for the project, along with setting developer standards.

Furthermore, he led the effort in resource management, overseeing aspects such as the procurement of essential resources, including the management of Amazon infrastructure payments. Eng. Mehari dedicated significant effort to the recruitment process, conducting interviews and identifying future team members. It's worth noting that as of May 19th, just two weeks before the conclusion of the internship, most team members were not yet onboarded.

Consequently, there was continued involvement between us and Eng. Mehari beyond the internship period.

The genesis of the project stemmed from Eng. Mehari's conceptualization, and regular meetings with him at the helm were convened to evaluate progress and chart the project's course.

Subsequently, we assumed the responsibility of conducting extensive research to identify the most suitable technologies, design an appropriate architecture, document it, and execute the project's implementation.

I.2.2 Target Audience

The *SHIDA* project is meticulously designed with a specific target audience in mind: individuals who share a profound passion for literature and possess a keen interest in critical thinking, knowledge sharing, and community engagement. While the project initially drew inspiration from the profile of an ardent reader, it extends its appeal to a broader community of like-minded individuals who value intellectual growth and social connections.

1. **Book Enthusiasts:**
   - **Readers of All Ages:** The platform welcomes readers of diverse age groups, including young adults, working professionals, and retirees, who share a common passion for books and knowledge.
   - **Frequent Readers:** It caters to individuals who engage in extensive reading, whether for leisure, educational purposes, or professional development.
2. **Knowledge Enthusiasts:**
   - **Critical Thinkers:** *SHIDA* appeals to those with a penchant for critical thinking, encouraging intellectual discussions and debates.
   - **Knowledge Contributors:** It provides a space for users to contribute their expertise, insights, and experiences to a broader community.
3. **Community Seekers:**
   - **Social Engagement:** The platform is ideal for those who seek social interactions and the formation of connections with like-minded individuals.
   - **Online Communities:** It offers features such as reader clubs and chat rooms, fostering a sense of belonging and camaraderie.

4. **Explorers of Diverse Interests:**
   - **Multifaceted Interests:** *SHIDA* accommodates individuals with diverse interests beyond literature, including travel, e-commerce, and learning.
   - **Lifelong Learners:** It caters to lifelong learners who are curious about exploring new horizons and gaining knowledge in various domains.

By understanding and catering to the unique preferences and aspirations of this diverse target audience, the *SHIDA* project aims to create a vibrant and inclusive digital ecosystem. This ecosystem fosters the exchange of ideas, the celebration of literature, and the pursuit of personal growth and connections within a community of passionate individuals.

I.2.3 Product Structure

The *SHIDA* product's structure embodies the multifaceted nature of its services. To provide a comprehensive understanding, we can break down its key components, sorted by priority, as follows:

1. **Digital Library:** A comprehensive collection of e-books and audiobooks, available for purchase or subscription, offering diverse reading options. Key features include:
   - **Community Engagement:** Fostering a sense of community through **Book Clubs** and **Talk Rooms**. These features enable group reading, discussions, book sharing, and live voice-based conversations, enhancing the communal experience.
   - **Reader Clubs:** Users can create or join clubs, connecting with like-minded individuals, organizing group reading, sharing book reviews, and recommending literature. Clubs also encourage friendly competition, strengthening community bonds.
   - **Writer Clubs:** Exclusive communities that promote creativity and collaboration among writers, facilitating feedback and creative development. These clubs establish a unique relationship with the publishing company, streamlining the process from draft to published book.
   - **Talk Rooms:** Dynamic spaces for real-time voice-based discussions. Users can create rooms, initiate conversations, and participate in live discussions.

- **Storefront Reviews:** A platform for users to leave reviews and ratings for purchased books, enhancing user interaction and feedback.

2. **E-Commerce:** This section is dedicated to providing exclusively designed *SHIDA* Collection merchandise items like mugs and shirts, as well as physical versions of books.

3. **Knowledge Hub:** This repository houses an array of e-learning articles authored by verified academics across various disciplines. Key functions include:
   - **Academic Expertise:** Content is exclusively created by verified academics, ensuring in-depth insights and reliability.
   - **Diverse Topics:** Articles span diverse categories, offering users an opportunity to explore a wide range of subjects.
   - **Interactive Features:** Users can interact with articles through comments, likes, and subscriptions, fostering a vibrant learning community.

4. **Villages Wiki:** This service allows users to access and contribute stories about villages in Eritrea and later expand to other African countries. Verified users can share truthful information about even the smallest villages, enriching cultural and historical knowledge.

5. **Matchmaking:** Designed to foster connections among users with similar interests, this service offers features like exclusive rendezvous options and direct messaging to facilitate meaningful interactions.

6. **Charity Hub**: A dynamic page where users can view a list of the latest and highest donors, along with the outcomes and impact of their generous donations, fostering a spirit of philanthropy and community support.
   - **Promotion of Donations**: Promoting donations that contribute to the education of African children and other charitable causes.
   - **User Gift Exchange**: Allowing users to send gifts to each other, fostering a culture of generosity and appreciation within the community.

7. **Creativity Hub**: Partnering with artists to showcase and sell photography, digital and physical artwork, animations, and games, promoting creativity and artistic expression within the platform.

8. **Money Transactions:** A user-friendly platform for efficient and cost-effective global money transactions, with reduced fees and top-notch security measures.

9. **Travel Packages:** An all-inclusive travel platform providing personalized travel experiences, including flights, accommodations, transportation, and activities, all with a focus on affordability and convenience.

I.2.4 Actors

The following context diagram in Figure 2 provides a high-level overview of the system and its internal and external entities, to help define project scope.

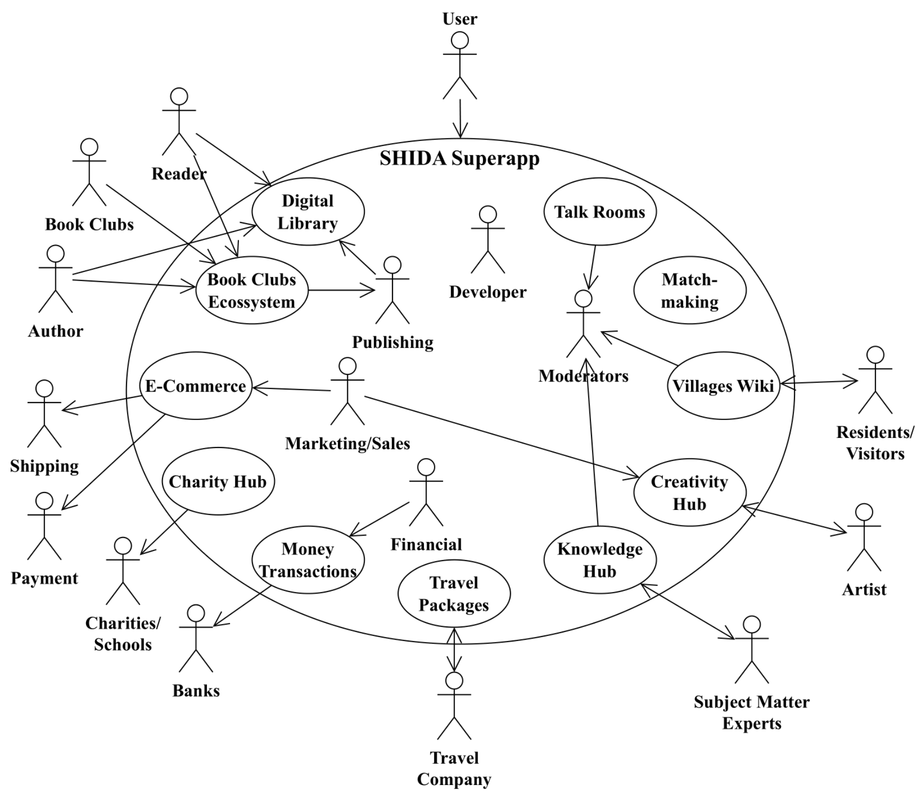


Figure 2: Context diagram for the *SHIDA* superapp.

II State-of-the-Art

In this chapter, we embark on a comprehensive exploration of the current technological landscape relevant to our project. By delving into the state-of-the-art, we aim to identify the most suitable technologies, frameworks, and practices that align with our project's objectives. This analysis provides a foundation for informed decision-making, ensuring that we leverage the latest advancements to create a cutting-edge and efficient solution.

Sources for relative popularity of most of the technologies mentioned in the following sections are obtained from Stack Overflow (2023), their latest survey as of writing.

II.1 Programming Languages

The selection of programming languages is a critical decision, especially in the context of the *SHIDA* project, which prioritizes code reuse and a low entry bar. To meet the system's scalability requirements, a modular architecture consisting of multiple independent parts is essential, which leads to the need of middleware like RPC (*Remote Procedure Call*) for inter-language communication.

Java, part of the mature Java family, is known for its platform independence, bytecode compilation, and object-oriented nature. Scala, Groovy, and Kotlin, while running on the JVM (*Java Virtual Machine*), introduce modern features. Kotlin, in particular, gained prominence as Google's preferred language for Android development.

Python and Ruby, dynamic and high-level languages, are recognized for their simplicity and readability. Python's large standard library and Ruby's elegant syntax make them suitable for web development, with Python excelling in rapid development.

The language C# and .NET, the platform built around it, are Microsoft's offerings for Windows applications and web services. While, in the past, .NET Framework and .NET Core differed in platform support, .NET version 5 in 2020 unified them, streamlining development. For certain cases there is also sister language F#, which employs a functional approach instead of being object-oriented.

JavaScript, integral to web development, faces challenges like its sometimes unpredictable dynamic typing. TypeScript, a superset, introduces static typing and advanced tooling for improved code quality. However, it adds a transpilation step, potentially complicating development.

PHP, a server-side scripting language, is widely used, notably in "website builders" like WordPress. Despite criticisms of inconsistency and security issues, modern versions and frameworks keep evolving to address these concerns.

C and C++ are venerable programming languages that remain powerful tools for systems programming. However, their use in modern web backends has diminished due to the rise of higher-level languages that offer greater developer productivity and flexibility. Such is the case of the Rust language, valued for its memory safety and performance, excels in server-side development thanks to its rock-solid stability and predictability.

Golang stands out with its focus on simplicity and built-in concurrency support, making it a compelling choice for developing scalable applications.

Dart, almost exclusively paired with the Flutter framework, targets cross-platform applications. Elixir focuses on concurrency and distributed systems, while Elm is a purely functional language made for robust web interfaces. These have limited applicability due to their smaller ecosystem and challenges in finding proficient developers and specific use cases.

The increasing emphasis on type systems has transformed the landscape of modern programming languages. Type systems provide a structured framework for defining and checking data types, ensuring code reliability and preventing runtime errors. While statically typed languages like Java and C# have long embraced type systems, their importance has permeated dynamically typed languages like Python and JavaScript. The introduction of type annotations and type inference mechanisms in these languages has significantly enhanced their type safety, making them more robust and maintainable. This growing adoption of type systems across programming paradigms underscores their fundamental role in ensuring code correctness and improving developer productivity.

Furthermore, the introduction of async/await patterns in languages like JavaScript, Python, C#, and Java marks a significant advancement in asynchronous programming, significantly simplifying development of efficient code. By eliminating the need for nested callbacks, these patterns enhance code readability and maintainability, making asynchronous programming more accessible and manageable.

II.2 Technology Stacks

Choosing the right technology stack is pivotal in application development, influencing factors from development efficiency to scalability.

A technology stack is a strategically chosen collection of software components that together provide the necessary foundation for building, deploying, and maintaining a software application. Key components include programming languages, frameworks, databases, environment, infrastructure and tools, each playing a distinct role in shaping the application's capabilities and performance.

In this context, the term “stack” refers to the hierarchical arrangement of software components, mirroring the multi-tiered architecture model in software

engineering, as described in Microsoft Learn (2014). This layered architecture fosters modularity, maintainability, and scalability by enforcing loose coupling between layers and strong cohesion within layers, effectively encapsulating concerns and abstracting away implementation details.

II.2.1 "Frontend" and "Backend"

In the current paradigm of software development, two primary domains exist: frontend and backend.

Backend development encompasses server-side components, including database management, security implementation, and application logic orchestration. It forms the backbone of the application, handling data processing, business rules, and communication with external systems.

Conversely, frontend development focuses on crafting user interfaces using markup, styles and scripting.

This division aligns seamlessly with the client-server model, fostering specialization and collaboration among developers. Full-stack developers possess expertise in both domains, bridging the gap between frontend and backend development.

II.2.2 History of Tech Stacks

The history of open-source tech stacks reveals evolving preferences, developer experience and specializations. The LAMP stack (Linux, Apache, MySQL, PHP/Python/Perl), described in Amazon Web Services (2022), was an early favorite. Later, MEAN (MongoDB, Express.js, Angular, Node.js) and MERN (React replacing Angular) gained prominence for full JavaScript development. Both of these and other stacks built around full-stack frameworks like RoR (*Ruby on Rails*), Laravel and Django, known for their "convention over configuration" and "batteries-included" approach, remain popular. One such example is the TALL[1] stack (Tailwind CSS, Alpine.js, Laravel and Livewire).

JAMstack, an acronym for JavaScript, APIs (*Application Programming Interface*), and Markup, emerged as a popular choice for building simpler websites like blogs and documentation. Its flexibility in markup languages, CI (*Continuous Integration*)/CD (*Continuous Delivery*) integration, and reduced maintenance costs make it an attractive choice for quick and easy web development.

[1] https://tallstack.dev/

More recently, stacks like T3[2] (Next.js, Prisma, tRPC) follow increasing prioritization of typesafety as a solution to reduce the amount of testing necessary. Simultaneously, BETH[3] stack (Bun, Elysia, Turso, HTMX) addresses criticisms such as Gross (2022), focusing on performance and innovation, which are mostly inspired in restoring the initial definition of REST (*Representational State Transfer*) that was drifted away from during the last decade's advancements.

II.3 The Web Browser

Web browsers are the gateways to the vast expanse of the WWW (*World Wide Web*). They serve as the interpreters that translate the intricate codes and markup languages, transforming them into visually appealing and interactive web pages. Delving into the intricacies of web browsers is an essential undertaking for web developers, as it equips them with the knowledge and skills to craft websites that seamlessly adapt to the diverse landscape of browsers.

The evolution of the WWW from static, read-only pages (Web 1.0) to dynamic, user-generated content (Web 2.0) and the upcoming semantic, decentralized and AI-driven Web 3.0 reflects technological advancements and changing user needs.

HTML (*Hypertext Markup Language*), the backbone of the web, has undergone a remarkable transformation since its inception as a means for displaying simple text documents. The current iteration, HTML5, represents a significant advancement, introducing a wealth of multimedia elements and standardized APIs that have revolutionized the way web applications are built. These enhancements have empowered developers to craft immersive and interactive web experiences, pushing the boundaries of what was previously possible with traditional web applications.

Notable browser features and APIs that have spearheaded this evolution include PWAs (*Progressive Web App*), Web Components, Web Storage, WebGPU, WebRTC, WebAssembly, Web Transitions, and the Dialog element. Each of these innovations has played a pivotal role in shaping the modern web landscape, enabling developers to create sophisticated web applications that deliver rich user experiences across a wide range of devices.

[2] https://create.t3.gg/
[3] https://github.com/ethanniser/the-beth-stack

II.3.1 Progressive Web Apps

PWAs (Progressive Web Apps) are a modern web technology that enables developers to create web applications that offer a native app-like experience.

PWAs are designed to be installable, fast, reliable, and engaging, even when offline. They can work offline, even when the user is not connected to the internet, which is a major advantage over traditional web applications. Additionally, PWAs can send push notifications to users, even when they are not using the app.

One of the key components of PWAs is the web manifest. The web manifest is a JSON (*JavaScript Object Notation*) file that provides information about the application, such as its name, description, icons, and background color. The web manifest also defines the PWA's offline capabilities, including which pages can be cached and how the PWA should handle network connectivity.

Another key component of PWAs are service workers. They are background scripts that run independently of the user's browser. This is what allows them handle tasks such as caching content, intercepting network requests, and providing offline functionality.

PWAs can also be installed on the user's home screen, which makes them look and feel like native apps, and this can increase user engagement and retention. Moreover, PWAs are often faster than traditional web applications, thanks to caching and other techniques, and this can improve the user experience, especially on slower devices. PWAs are also designed to be reliable, even in low-bandwidth environments, and this is especially important for apps that are used in areas with poor internet connectivity.

However, there are some limitations to PWAs that developers need to be aware of. For example, PWAs do not yet have access to all of the same APIs as native apps, which can limit their functionality in some cases. Additionally, not all browsers support all of the features of PWAs, which can create a fragmented experience for users. Finally, developing PWAs requires additional effort compared to traditional web applications, which may be a barrier for some developers.

Nonetheless, prominent companies, including Twitter, Spotify, Pinterest, Telegram, Instagram, AliExpress, Tinder, and Lyft, have successfully implemented PWA features. As documented in Hodakovskis (2019), the companies have claimed improved user engagement, higher conversion rates, and enhanced performance in low-bandwidth scenarios.

It's worth noting that these PWAs are not intended to replace their native counterparts; instead, they are often referred to as "Lite" versions, coexisting to offer users a more versatile experience.

II.3.2 Performance Metrics

Browser performance is critical for providing a seamless user experience. It directly impacts user satisfaction, website engagement, and ultimately, business outcomes. Sluggish or unresponsive websites can lead to increased bounce rates, reduced page views, and decreased conversions.

Nowadays, it can be easily measured and audited using tools like the Core Web Vitals API and Google Lighthouse, with debugging facilitated by developer tools. The categories measured by these tools focus on Performance, Accessibility, Best practices, SEO (*Search Engine Optimization*), and PWA support.

Within the Performance category, the important metrics[4] taken into account are FCP (*First Contentful Paint*) and LCP (*Last Contentful Paint*) (time it takes for elements to load), Speed Index (loading time of the viewport), Total Blocking Time and Time to Interactive (time it takes for the browser to reliably respond to user interaction) and Cumulative Layout Shift (amount of unexpected layout shifts that occur during page loading).

II.3.3 Different "Flavors"

Modern browsers use rendering engines such as Chromium's Blink (used by Chrome and several other browsers), Mozilla's Gecko (used by Firefox), and Apple's WebKit (used by Safari). These engines are responsible for rendering web content according to the latest web standards. Figure 3 illustrates the main flow of the WebKit engine:

[4] https://web.dev/explore/metrics

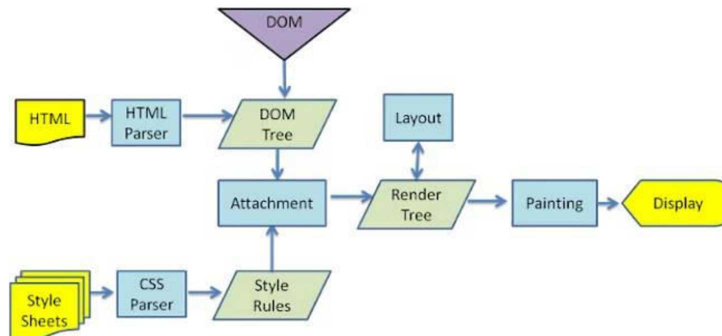


Figure 3: Flowchart depicting the core processes within the WebKit engine (source: Garsiel and Irish (2011))

V8, SpiderMonkey, and JavaScriptCore are the core JavaScript engines behind popular browser engines. V8 powers Blink, SpiderMonkey fuels Gecko, and JavaScriptCore drives WebKit. These engines are pivotal in executing JavaScript code efficiently in browsers, and understanding their nuances is vital for web developers aiming for optimal performance and compatibility.

The browser landscape is dominated[5] by Chromium-based browsers, notably Google Chrome, Microsoft Edge and Opera, while Mozilla Firefox remains prominent for its focus on user privacy and open-source principles. Additionally, Safari's unique position within the Apple ecosystem and its adherence to Apple's strict platform guidelines have contributed to both its substantial market share and its relative isolation from other browsers in terms of API compatibility.

The diversity of browser engines is a double-edged sword. It drives competition and innovation but also introduces compatibility challenges. Developers must navigate engine-specific quirks to ensure cross-browser compatibility: notable tools that helps with this aspect is the "Can I use?" website[6] and the Browserlist configurations[7]. As web standards evolve, understanding engine variations is crucial for crafting consistent and compatible web applications.

II.4 Application Layer

APIs and services are often used interchangeably, but they have distinct roles in software architecture.

[5]https://gs.statcounter.com/browser-market-share
[6]https://caniuse.com/
[7]https://browsersl.ist/

In modern software architecture, APIs and services play distinct but complementary roles. APIs serve as the standardized interface that defines how applications interact with each other, while services encapsulate the business logic that fulfills specific tasks or processes.

APIs act as intermediaries between the presentation layer (user interface) and the business logic layer, enabling seamless communication between different components of an application. They define the rules, procedures, and protocols for interacting with the application's functionalities, allowing developers to integrate these functionalities into their own projects without delving into the underlying implementation details. This promotes modularity, flexibility, and maintainability, making APIs crucial for developing complex, distributed systems.

Services, on the other hand, encapsulate the business logic that fulfills specific tasks or processes. They are the implementation behind the APIs, providing the actual functionalities that are exposed through the API. Services are often designed as independent, self-contained components that can communicate with each other through APIs to achieve a common goal. This modularity promotes code reusability, scalability, and fault tolerance.

To prioritize seamless integration and interoperability with external systems, teams may adopt an API-first approach, placing a greater emphasis on API development over data modeling or user interface design.

This explanation of API design principles was initially inspired by the comprehensive resources found under Lam *et al.* (2023).

II.4.1 Distribution Patterns

There are multiple different ways to distribute components or data across nodes or environments, each with distinct characteristics and suitability for specific use cases.

The client/server architecture lays the groundwork for distributed systems, facilitating communication between a client and a server. This pattern encompasses a wide range of interactions, from simple data retrieval to complex real-time data exchange.

The traditional client/server model, where a client initiates a request and has to wait for a response from the server, can lead to tight coupling and performance bottlenecks. To address these limitations, more sophisticated client/server architectures have emerged, utilizing asynchronous communication and message brokers to decouple components and enhance scalability. These ad-

vancements have solidified client/server as a versatile and scalable architectural pattern for modern software systems.

The P2P (*Peer-to-peer*) pattern eliminates the central server and instead allows all clients to communicate directly with each other. This pattern is highly scalable and fault-tolerant, as the network can continue to operate even if individual nodes fail. P2P is well-suited for applications that require distributed file storage, content sharing, and decentralized data management.

Serverless computing abstracts away the management of servers and infrastructure to cloud providers, allowing developers to focus on writing code without worrying about provisioning and maintaining servers. This pay-per-use model can reduce operational overhead and costs. Serverless computing is well-suited for event-driven applications, batch processing, and tasks that require on-demand scalability.

Other options for distribution include fog computing, which extends cloud computing closer to the edge of the network, bringing computing resources closer to end-users and data sources, and CDNs (*Content Delivery Network*), which strategically cache content at geographically distributed locations to reduce latency and improve content delivery efficiency.

II.4.2 Structural Patterns

Structural patterns represent ways to organize and compose components within a single node or environment.

As one such case, the Monolith architecture places all code in a single codebase, making it easy to develop, maintain, and deploy for small to medium-sized applications. The unified codebase simplifies development and debugging, and is well-suited for applications with a cohesive code structure and limited complexity.

SOA (*Service-Oriented Architecture*), on the other hand, organizes software into independent services that communicate through well-defined interfaces. This modular approach promotes loose coupling, reusability, and scalability, making it well-suited for complex enterprise applications. Each service encapsulates a specific business capability, enabling independent development, deployment, and scaling.

Microservices architecture takes SOA to a more granular level, where each service is responsible for an atomic business capability. This approach further enhances flexibility, scalability, and independent development and deployment. Microservices are self-contained, independently deployable, and communicate

through lightweight protocols, enabling rapid development and adaptation to changing requirements.

Related to serverless computing, edge functions enhance application performance by executing functions at the network edge, closer to the end-users or data sources. This reduces latency and improves responsiveness for real-time applications, such as IoT devices and streaming video services. Edge functions are becoming increasingly relevant as the IoT (*Internet of Things*) and edge computing technologies evolve.

To address potential congestion and improve scalability, performance, and maintainability, CQRS (*Command Query Responsibility Segregation*) (Command Query Responsibility Segregation) emerges as an architectural pattern. This approach separates the handling of data updates (commands) from data retrieval (queries), optimizing each side for its specific purpose.

In some cases, it may be beneficial to combine elements from different patterns to create a hybrid architecture that leverages the strengths of each approach. For instance, a microservices architecture could be integrated with a monolith for certain components that require tight coupling or a high degree of control. Hybrid architectures can offer increased flexibility and scalability, but they also introduce additional complexity and management overhead.

II.4.3 Communication Patterns

APIs can follow certain communication patterns depending on their use case. These patterns define how different components of an application interact with each other through the API.

The Request/Response model is a fundamental communication pattern used in various protocols and technologies. It is a simple and straightforward approach for basic interactions between a client and a server. However, it can become a bottleneck if the server receives a high volume of requests, as the client must wait for the server to respond before proceeding.

In contrast, the Event-driven model, often implemented as Publish/Subscribe, fosters loose coupling by allowing asynchronous communication through message topics, ideal for real-time applications and event-driven systems.

Finally, the Push/Pull model is a mechanism for data transfer where data is either actively pushed from the server to clients or pulled by clients on-demand.

II.4.4 Communication Protocols

The realm of communication protocols offers a diverse array of tools for facilitating data exchange between application components. Each protocol boasts distinct features and caters to specific use cases, making the choice of protocol a critical decision in application development.

SOAP (*Simple Object Access Protocol*) stands as a cornerstone of enterprise-grade communication, renowned for its maturity and comprehensiveness. Its adherence to XML-based (*Extensible Markup Language*) information exchange ensures strict data integrity and security, making it a natural choice for scenarios demanding robust data protection.

However, SOAP's intricate structure can introduce overhead and complexity, particularly in resource-constrained environments. This challenge paves the way for the emergence of RESTful architecture.

The RESTful architecture embraces simplicity and efficiency, adhering to HTTP (*Hypertext Transfer Protocol*) methods and promoting statelessness and a uniform interface. This approach makes REST ideal for web services and CRUD (*Create, Read, Update, Delete*) operations, offering a lightweight and performant solution.

While REST's resource-oriented approach simplifies data access, it can pose challenges in handling complex data relationships and retrieving specific data subsets. This limitation gives rise to GraphQL, a query-centric protocol.

GraphQL introduces a paradigm shift in data retrieval, empowering developers to specify the exact data they need through a query language. This approach eliminates over- and underfetching, reducing network overhead and roundtrips, resulting in faster responses.

Despite GraphQL's flexibility, it comes with the potential for increased complexity on the client side and potential abuse if not properly safeguarded. This challenge underscores the need for protocols tailored to specific communication requirements.

gRPC emerges as a modern and high-performance protocol, leveraging Protocol Buffers (Protobuf) for efficient binary encoding. This approach delivers exceptional performance, making it well-suited for memory and performance-critical applications, particularly in microservices architectures.

WebSockets provide a bidirectional communication channel for real-time web interactions, enabling continuous and uninterrupted data streaming. This fea-

ture makes WebSockets ideal for applications that demand real-time data updates and collaboration, such as live chat.

Webhooks, characterized by their event-driven nature, utilize HTTP callbacks for asynchronous communication. This mechanism facilitates timely and efficient event handling, making Webhooks a valuable tool for notifying systems when specific events occur.

AMQP (*Advanced Message Queuing Protocol*) and MQTT play a pivotal role in distributed applications, particularly in IoT environments, by providing reliable asynchronous communication. Its utilization of brokers facilitates the exchange of messages between producers and consumers, ensuring robust and reliable data transfer.

II.4.5 API-related Tools

API gateways serve as centralized entry points for accessing APIs, acting as intermediaries between clients and backend services. They provide a single interface for managing API traffic, implementing security measures such as authentication and authorization, and enforcing rate limiting to prevent abuse. Additionally, API gateways can offload tasks like routing, load balancing, and protocol translation, reducing the burden on backend servers and improving overall performance.

Message brokers like RabbitMQ are a type of middleware that facilitates asynchronous communication between applications using the pub/sub pattern. They act as central hubs for message exchange, allowing publishers to broadcast messages without knowing who will consume them, and subscribers to express interest in specific topics or types of messages.

The OAS (*OpenAPI Specification*), formerly known as Swagger, provides a machine-readable format for describing REST APIs. This standardized format enables developers to create comprehensive API documentation, facilitating collaboration and understanding among team members and external consumers. Additionally, OAS serves as a foundation for generating code stubs, client libraries, and API documentation, promoting language-agnostic APIs that can be easily integrated into various development environments.

API testing is an integral part of the software development lifecycle, ensuring that APIs meet functional requirements, perform well under load, and maintain security and reliability. A range of API testing tools exists to address various aspects of API quality, including, for functional testing: tools like Postman and

SoapUI enable developers to test API functionality by sending requests and validating responses against expected behavior.

For performance testing, tools like JMeter and Gatling can simulate high traffic scenarios to evaluate API performance, identify bottlenecks, and ensure the system can handle peak load.

Tools like OWASP ZAP and Burp Suite do security testing by scanning APIs for vulnerabilities, such as SQL (*Structured Query Language*) injection and cross-site scripting, to ensure data protection and prevent unauthorized access.

By employing these API testing tools, developers can proactively identify and address potential issues, ensuring that APIs are robust, reliable, and meet the expectations of consumers.

II.5 Presentation Layer

The design of the presentation layer in software development involves the use of application patterns, which are reusable solutions to common design challenges. These patterns aid developers in creating more efficient, scalable, and maintainable applications, particularly in the context of web development where they describe strategies for structuring and rendering web applications.

II.5.1 Framework Architectures

Architectural patterns, such as MVC (*Model-View-Controller*), MVVM (*Model-View-ViewModel*), MVP (*Model-View-Presenter*) and VIPER (*View-Interactor-Presenter-Entity-Router*), offer solutions to design challenges, enhancing code flexibility, reusability, and maintainability. The choice of pattern depends on the specific requirements of the application.

Most modern front-end frameworks employ component-based architectures, which are a software development methodology that organizes code into reusable and modular components. Each component is responsible for a specific task or piece of functionality, and they can be easily combined to create complex applications.

In other cases, template-based frameworks define the structure of the view using templates, which are HTML documents that contain special placeholders for dynamic content. JavaScript is then used to manipulate the content of the templates, dynamically generating the final UI (*User Interface*).

Finally, declarative frameworks take a different approach, using declarative syntax to describe the desired state of the UI. The framework then handles the low-level details of rendering the UI to the screen. This can make declarative frameworks more concise and expressive, but they can also be less intuitive for beginners to learn.

II.5.2 Application Structure

There are two main approaches to structuring web applications: MPAs (*Multi-Page Application*) and SPAs (*Single-Page Application*). MPAs involve separate HTML files for each page, resulting in full page reloads during navigation. While simpler to develop, they can have slower initial load times.

On the other hand, SPAs interacts with the user by dynamically rewriting the current page rather than loading entire new pages from the server. In a SPA, the page does not reload during user navigation, and content is updated asynchronously by JavaScript. SPAs offer a more fluid and responsive user experience by minimizing the need for full-page reloads. However, SPAs can be more challenging to develop and maintain due to their reliance on JavaScript for state management and the significant bundle size they require to be transferred to the client.

While still a relatively uncommon architectural approach, microfrontends have emerged as a promising solution for building large-scale web applications. By breaking down the application into smaller, independent frontend modules, microfrontends offer modularity, scalability, and the ability to leverage the best features of both SPAs and MPAs. However, the high complexity of managing and coordinating these distributed modules has hindered their widespread adoption.

In domains that rely on organic traffic like e-commerce and content-driven websites, effective SEO is crucial, especially for web crawlers. It involves various strategies, including creating sitemaps, crafting well-structured `robots.txt` files, optimizing meta tags, implementing structured data markup, enhancing page speed, prioritizing mobile optimization, producing high-quality content, building backlinks, utilizing analytics, and ensuring HTTPS (*Hypertext Transfer Protocol Secure*) implementation.

II.5.3 Rendering Strategies

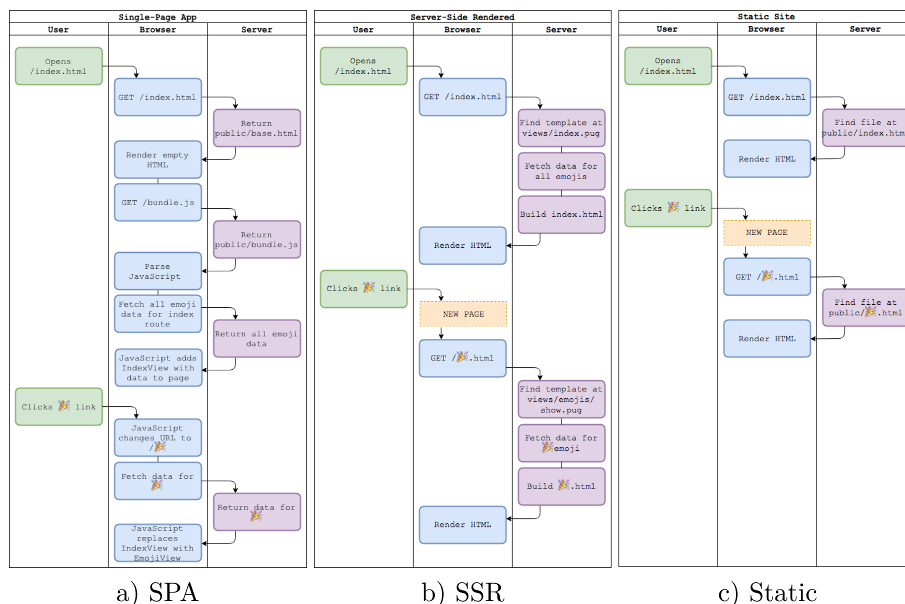
SPAs commonly employ CSR (*Client-Side Rendering*), which refers to the practice of rendering web pages on the client's side (in the user's browser) rather

than on the server. In the context of SPAs, CSR involves loading a minimal HTML page and then using JavaScript to dynamically update and render the content on the client side, usually by taking advantage of front-end frameworks like React and Angular.

In contrast, SSR (*Server-Side Rendering*) generates fully populated HTML pages on the server on each request, providing fast initial loading but with higher server resource costs.

SSG (*Static Site Generation*), on the other hand, pre-builds HTML files at build time, offering excellent performance benefits and security for the cost of interactability with the website.

The following Figure 4 illustrates the process behind loading resources on each of these models.



a) SPA b) SSR c) Static

Figure 4: Sequence diagrams for different models for rendering a website

Leveraging Streaming HTML, also known as SSE (*Server-Sent Events*), allows the server to send data to the client as soon as it's available, enabling real-time updates and dynamic content without constant client requests.

Citing Next.js (2023), *"Streaming allows the breaking down of the page's HTML into smaller chunks and progressively sending those chunks from the server to the client. This enables parts of the page to be displayed sooner, without waiting for all the data to load before any UI can be rendered. Streaming*

works well with React's component model because each component can be considered a chunk."

As the demands of web applications have evolved, more sophisticated rendering strategies have emerged to address the limitations of conventional approaches. ISR (*Incremental Static Regeneration*), for example, combines the benefits of SSG with dynamic content updates, allowing specific pages or content blocks to be revalidated and updated at intervals.

In contrast, Isomorphic JavaScript combines SSR with client-side interactivity, providing a faster initial page load and improved SEO. The server generates fully populated HTML pages, while client-side JavaScript handles subsequent interactions.

Isomorphic JavaScript excels in applications that demand seamless transitions from SSR to dynamic interactivity, particularly those with frequent user interactions and data handling complexities. ISR, on the other hand, shines in applications with frequently visited content and manageable dynamic updates, where maintaining SEO and reducing initial load times are paramount.

Hydration, the transformation of pre-rendered HTML components into live, interactive elements on the client-side, plays a pivotal role in achieving a seamless transition from initial SSR to dynamic interactivity.

Numerous hydration techniques, such as Partial Hydration, Progressive Hydration, and Universal Rendering, further elevate rendering performance in web applications. They selectively target specific parts of the webpage, eliminating the need for re-rendering the entire page with every interaction.

More recently, Server Components represent an innovative approach that introduces a hybrid rendering strategy. It empowers specific components to be rendered on the server, while others reside on the client, effectively leveraging the benefits of both SSR and CSR. This approach has the potential to optimize performance and resource utilization in complex applications.

II.5.4 Data Loading

Client-side data fetching has become an indispensable tool for modern web development, enabling developers to create responsive and dynamic applications that seamlessly interact with APIs, databases, and services. The advancements in data fetching techniques have revolutionized user experiences and paved the way for sophisticated web applications.

Tools and technologies

AJAX (*Asynchronous JavaScript and XML*) emerged in the early 2000s, introducing asynchronous data exchange and eliminating the need for full page reloads. Today, modern AJAX applications primarily utilize JSON data, leveraging JavaScript's fetch API, the `async` and `await` keywords to build real-time, dynamic web experiences.

The Fetch API marks a significant turning point in handling network requests and server interactions. It provides a cleaner, promise-based syntax for fetching resources like JSON data, offering greater flexibility and control over request options. This facilitates error handling, response processing, and the creation of responsive, data-driven web applications.

Promises, a built-in JavaScript feature, introduce a more structured approach to handling asynchronous code. They represent values that may not be immediately available but will become so in the future. Callbacks can be attached to pending, fulfilled, or rejected states, streamlining the management of asynchronous operations.

Methods

The choice of data fetching method depends on the specific requirements of the application:

While polling serves as a fundamental data fetching method, it can be inefficient for applications with infrequent updates, because of the overuse of resources. A more efficient approach for such scenarios is to combine a lower polling rate with server-side caching, which involves storing requested data on the server for a specified duration. This allows subsequent requests to retrieve the data from the cache, reducing the number of network requests and improving overall application performance.

For real-time applications, where data updates occur at a rapid pace, polling is also often inadequate because of the possible latency and network overhead. Long polling, a variation of polling, mitigates this issue by keeping the connection open between the client and server. However, it may still introduce latency if updates are not frequent.

SSE and WebSockets offer more efficient solutions for real-time data fetching. With SSE, the server actively pushes data to the client as it becomes available, eliminating the need for constant polling or long polling. WebSockets, on the other hand, establish a persistent connection between the client and server, enabling real-time data exchange with minimal latency.

II.6 The JavaScript Ecosystem

The JavaScript ecosystem is a dynamic and vast collection of tools, libraries, and frameworks crucial for modern web development. JavaScript's versatility is evident in its various runtimes, with Node.js being by far the most popular for server-side scripting.

Bun, released recently as of September 8, 2023 (Sumner, Partovi and McDonnell (2023)), is positioned as a comprehensive toolkit addressing performance and complexity issues in the JavaScript ecosystem.

With their more special use cases, cloud-specific runtimes like AWS Lambda and edge runtimes such as CloudFlare Workers can be employed to optimize content delivery.

Package managers like npm, Yarn, and pnpm simplify dependency management, sourcing libraries from registries like the npm registry.

Associated to these, workspaces enhance project organization, integrating seamlessly with monolithic repository tools like Lerna and Rush, further explored in Section II.14. The choice among package managers depends on factors like installation speed and storage consumption.

ES Modules and CommonJS represent contrasting approaches to module organization in JavaScript. As of 2023, CommonJS, the traditional module system, is gradually losing ground to ES Modules, the emerging standard. While CommonJS relies on additional tooling and directory-based organization, ES Modules offer native browser support, tree shaking for efficient code bundling, and better support for asynchronous operations. This shift reflects the preference for a more streamlined and efficient module system.

Build tools and bundlers, such as Babel, SWC, Webpack, Rollup, Vite, Parcel, and Turbopack, empower developers to optimize application performance and streamline development workflows. These tools transform and bundle code for efficient delivery, handle complex dependencies, and facilitate seamless integration with modern JavaScript frameworks.

II.6.1 Frameworks

The landscape of front-end development frameworks has evolved significantly, with legacy frameworks like jQuery, Backbone.js, and AngularJS paving the way for modern approaches. Today, a diverse range of frameworks, includ-

ing Angular, React, Vue.js, Svelte, Solid.js, Qwik, and others, offers distinct methodologies for building user interfaces.

Innovations like JSX in React and the Composition API in Vue.js have transformed the way developers interact with these frameworks. Svelte stands out with its compiler-based approach, while Solid.js prioritizes speed and efficiency by pioneering the reintroduction of Signals, a design pattern that been getting reintroduced into new versions of the most established frameworks, such as Angular's major version 16[8].

In the backend domain, Express.js, Koa.js, Fastify, NestJS, Hono.js, and Elysia.js cater to diverse backend development needs. Most focus on having better performance, while some like Express.js focus on being lightweight and intuitive, and NestJS focuses on being having a more complete set of features.

Metaframeworks, such as Next.js, Remix, and Nuxt, provide comprehensive solutions for routing, server-side rendering, and other advanced features. Angular, while not being strictly a metaframework, enforces a specific structure and toolset for building large-scale applications. Emerging metaframeworks, like SvelteKit, SolidStart, and Qwik City, complement their respective UI frameworks, offering streamlined development experiences.

SSGs simplify web development by transforming content and templates into pre-built pages, aligning with the JAMstack philosophy. Astro, a modern SSG, stands out for its speed and minimal JavaScript payload, making it ideal for content-driven sites. Most metaframeworks, while not exclusive to SSG, can be configured to work like one.

JavaScript has a rich ecosystem of core libraries that caters to diverse web development needs, empowering developers to build robust, scalable, and user-friendly applications. State management libraries like Zustand[9] ensure data consistency, schema validation libraries like Zod[10] enforce data integrity, routing libraries like Tanstack Router[11] simplify navigation, data fetching libraries like Tanstack Query[12] streamline data retrieval, and many more. This comprehensive toolkit empowers developers to create modern web experiences that meet user expectations by trusting someone else has laid the groundwork for them, and their open-source nature enables collaborative development and continuous improvement.

[8] https://angular.dev/guide/signals
[9] https://zustand-demo.pmnd.rs/
[10] https://zod.dev/
[11] https://tanstack.com/router/latest
[12] https://tanstack.com/query/latest

The JavaScript ecosystem continues to evolve, addressing performance, scalability, and developer experience, with frameworks and tools catering to diverse needs in web development.

II.7 Other Languages on the Web

Languages other than JavaScript face a unique challenge on the frontend: they don't natively run on web browsers (at least not without additional tools like WebAssembly). As a result, these languages typically rely on full-stack web frameworks that employ a combination of templating techniques with JavaScript to facilitate client-side interactions. This approach allows developers to harness the power of various programming languages for both server-side and client-side functionality in web applications.

The development of API endpoints and servers is typically handled using the same dedicated web frameworks that offer a wide range of tools and libraries to streamline the process. Modern web frameworks provide a valuable abstraction layer, enabling developers to focus on application-specific logic rather than the low-level intricacies of HTTP request and response handling. This abstraction is particularly evident in Python's Web Server Gateway Interface (WSGI) and Java's Servlet technology.

Examples of widely used non-JavaScript web frameworks include those under the .NET Stack (ASP.NET Core, Blazor), Python's robust Django and lightweight Flask, and PHP's comprehensive Symfony and straightforward Laravel.

II.8 Mobile Platforms

In the dynamic realm of mobile platforms, the choice of technology stack plays a pivotal role. While the historical decision revolved around native vs. hybrid approaches, the landscape has evolved with the rise of cross-platform frameworks like React Native, Xamarin, and Flutter, narrowing the performance gap (see Maggini (2019)).

The mobile OS (*Operating System*) landscape has been dominated by Android and iOS for the past decade, holding a 70/30 split respectively[13]. Harmony OS, developed by Huawei, emerged as a significant player in 2019, but is primarily focused on the Chinese market. These OS offer distinctive features

[13]https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

and target different user preferences, with Android prioritizing open ecosystem, device diversity, customization, and app abundance, while iOS emphasizes closed ecosystem, hardware-software integration, consistent design, and privacy/security focus.

Navigating the app stores, Apple's exclusive, stringent App Store contrasts with Android's open and diverse Play Store. They are key pieces in deploying any mobile application.

Fully native mobile app development creates apps tailored to its specific OS, using native languages and SDKs (*Software Development Kit*), resulting in optimized performance, native UI, and granular control. However, it demands in-depth knowledge and can be complex, requiring maintenance of a large codebase. Native IDEs (*Integrated Development Environment*) like Xcode and Android Studio aid development with comprehensive tools.

II.8.1 Navigation

The differences in browser routing compared to mobile navigation pose a challenge that developers of hybrid apps have to take into account.

Browser routing allows for a more complex and hierarchical navigation structure, with multiple levels of pages and subpages. This can be useful for websites with a large amount of content, but it can also make it difficult for users to find their way around the site.

Mobile navigation, on the other hand, is typically simpler and more streamlined, with fewer levels. This is because mobile screens are smaller and have less space to display navigation elements. Additionally, mobile users are often on the go and need to be able to navigate easily.

Browser routing is stateless because, according to the REST model, it does not need to track the user's current navigation state. Instead, the user's navigation state is represented by the URL in the address bar. When the user clicks on a link, the browser navigates to the URL specified in the link. The browser does not need to know anything about the user's previous navigation history to do this.

Mobile navigation, on the other hand, is not stateless. Mobile apps typically need to track the user's current navigation state in order to provide a good user experience. For example, when a user taps the back button in a mobile app, the app needs to know which screen to return to.

Deep linking seamlessly integrates web URLs (*Uniform Resource Locator*) within app URLs, enabling smooth transitions between mobile apps and web pages. Unlike web hyperlinks, mobile hyperlinks use custom URL schemes or intent URLs for precise control over app behavior, enhancing the user experience.

II.8.2 User Actions

Mobile app developers, also have to be aware of the differences between mobile gestures and mouse and keyboard actions, especially when designing cross-platform applications.

Mobile devices do not have a cursor, so hovering is not available. This means that UI should be designed in a way that can be interacted with without using it. For example, using buttons, tabs, and other tappable elements to allow users to navigate the app and accessing its features.

Moreover, the left mouse click is replaced with tap on mobile devices. This means that all of the tappable elements should be large enough and easy to tap. Too many tappable elements on the same screen should be avoided, as this can make it difficult for users to tap the element they want.

Similarly, the right mouse click is replaced with a long tap on mobile devices. This means that developers need to decide which actions, if any, to trigger with a long tap. For example, a long tap could be used to open a context menu, reveal additional options, or perform another action that is not available with a single tap.

Mobile devices support gestures such as swiping, pinching, and zooming. These gestures can be used to allow users to interact with the app in new and innovative ways. For example, a swipe gesture to scroll through a list, a pinch gesture to zoom in on a map, or a zoom gesture to enlarge the entire screen.

II.8.3 Hybrid Frameworks

Mobile application frameworks, including NativeScript, Ionic, Flutter, React Native, and Xamarin, offer diverse approaches to cross-platform development.

Flutter stands out for its adherence to Google's design principles and widget-based architecture. React Native, with its "learn once, write anywhere" philosophy, uses JavaScript and React for native-feeling apps. Xamarin, using C# and .NET, prioritizes authentic native experiences. Each framework has strengths – Flutter's performance, React Native's versatility, and Xamar-

in's .NET integration – coupled with challenges, such as Flutter's long-standing iOS animation issues[14] or React Native's pixel-perfect design consistency struggles[15].

In the rapidly evolving landscape of mobile application development, the choice between these frameworks hinges on project requirements, development speed, and the desired user experience.

II.9 Desktop Platforms

Desktop platforms, designed for personal computers (PCs), include popular systems like Windows, Linux, and macOS.

Windows, the commercial OS by Microsoft, boasts a 70% market share with its effort for retrocompatibility and user-friendly interfaces.

Linux, free and open-source, is gaining traction among developers, known for its many flavors and portability.

MacOS, exclusive to Apple hardware, stands out for its elegant design and focus on productivity.

Each OS uses different libraries and drivers (e.g., DirectX for Windows, OpenGL for Mac, Vulkan for Linux), necessitating separate compilation for each.

Windows, although expensive, is widely compatible. Linux is free but can be challenging. macOS, exclusive to Apple, balances user-friendliness and design but is somewhat inflexible. Native desktop apps, written in the platform's language (e.g., C++, C#), excel in cases where performance and device feature access are key concerns.

Popular frameworks for developing desktop apps include Electron and Qt. Electron and Tauri, using web technologies, allow cross-platform development but may be more resource-intensive than other options. Qt, a C++ framework, prioritizes performance but requires more development effort.

React Native and Flutter, although designed for mobile, can develop native desktop apps.

[14] https://thomasmiddel.medium.com/flutter-its-poor-ios-performance-84ec1eacd235

[15] https://www.bam.tech/en/article/react-native-think-twice-before-fixing-your-components-height

Platform-independent GUI (*Graphical User Interface*) libraries[16] like Kivy for Python vary from lightweight to extensive, requiring manual programming support. For Java there is the older Swing and the newer JavaFX.

Over time, obsolete OS-locked frameworks like Microsoft's WPF and WinForms are replaced by modern, cross-platform, web-based alternatives thanks to advancements in web standards.

II.10 Data Layer

The data layer, responsible for storing, retrieving, and managing data in an application, encompasses components like databases, data access layers, and ORM (*Object-Relational Mapping*). Selecting an appropriate data layer involves considerations such as scalability, performance, security, ease of use, data type, budget, and community support.

RDBMS (*Relational Database Management System*) are prevalent, utilizing tables with rows and columns. They ensure data integrity through ACID (*Atomicity, Consistency, Isolation, and Durability*) transactions, crucial for reliable and complete database operations. Examples include PostgreSQL, known for extensibility, scalability, and support for advanced features; MySQL, valued for ease of use and performance; SQL Server, featuring an extension of SQL, Transact-SQL; and MariaDB, compatible with MySQL. SQLite, a lightweight embedded RDBMS, is portable and suitable for applications requiring a compact, serverless database.

Non-relational DBMS (*Database Management System*), also known as NoSQL databases, offer flexibility and scalability, accommodating diverse data types. Document databases, like MongoDB, store data in JSON or XML objects, ideal for complex data. Key-value databases, such as Redis, store data as key-value pairs, suitable for simple data storage. Columnar databases, such as Apache Cassandra, are adept at analytical workloads, while graph databases like Neo4j store data in graph structures, beneficial for complex relationships. Finally, time-series databases, like InfluxDB, are optimized for time-stamped data.

Database-as-a-service (DBaaS) in the cloud, exemplified by Firebase, AWS (*Amazon Web Services*) services (DynamoDB, RDS, Aurora, Redshift, MemoryDB), CloudFlare (D1, R2, KV), Supabase, PlanetScale, Neon, Fauna, and others, offer convenient access without managing infrastructure. While they

[16] https://en.wikipedia.org/wiki/List_of_platform-independent_GUI_libraries

can be user-friendly, cost-efficient, and scalable, they pose challenges such as data ownership clarity, potential vendor lock-in, and reduced control.

ORM simplifies data translation between a relational database and object-oriented programming languages, creating a virtual object database. ORM tools enhance development efficiency, provide abstraction layers, and offer performance optimizations such as caching and lazy loading. Developers can further optimize database queries using tools like database profilers.

II.11 Networking

The application layer, the highest level in the OSI (*Open Systems Interconnection*) model, facilitates network access for applications, while underlying layers manage networking functions like data transport and routing.

HTTP, a fundamental protocol, has evolved from HTTP/1.1 to HTTP/2 and the recent HTTP/3, enhancing web performance through features like multiplexing and reduced latency.

HTTPS, crucial for security, encrypts data using SSL (*Secure Sockets Layer*)/TLS (*Transport Layer Security*) certificates obtained from trusted CAs (*Certificate Authority*) like Let's Encrypt.

DNS (*Domain Name System*), a decentralized system, translates domain names to IP (*Internet Protocol*) addresses, essential for web navigation. To this extent, domain registrars, like Porkbun[17], CloudFlare's and AWS's, manage domain registrations.

Multimedia streaming, critical to major applications like Netflix and YouTube, relies on state-of-the-art protocols to deliver optimal user experiences. RTP (*Real-time Transport Protocol*) ensures real-time audio and video transmission, while adaptive streaming protocols like HLS (*HTTP Live Streaming*) and MPEG-DASH optimize quality and minimize buffering across devices and networks.

II.12 Security

Nowadays, the threat level on the web is high. The increasing sophistication of cyberattacks, proliferation of internet-connected devices and growing reliance on cloud computing challenge the security of users and systems alike.

[17] https://porkbun.com/

For web applications, security measures include CORS (*Cross-origin Resource Sharing*), controlling resource access; firewalls for network security; DMZs (*Demilitarized Zone*) to protect servers; and WAFs (*Web Application Firewall*), specifically safeguarding web applications against attacks such as DDoS (*Distributed Denial-of-Service*).

The OWASP (*Open Worldwide Application Security Project*) Top 10 lists critical web security risks, including broken access control, cryptographic failures, injection (XSS (*Cross-site Scripting*) included), and security misconfiguration. CVEs (*Common Vulnerabilities and Exposures*), categorized into OWASP threats, highlight publicly disclosed vulnerabilities. Defense against XSS involves content security policies, user input encoding, and regular software updates.

II.13 Server Environment, Deployment, and Hosting

Choosing the right server environment, deployment strategy, and hosting solution is crucial in the ever-changing field of web development. Traditional web hosting, offered by services like Linode or DigitalOcean, provides stability with options such as shared hosting, VPS (*Virtual Private Server*), and dedicated servers, each offering different levels of control and resource allocation.

Proxy servers and CDNs are pivotal for optimizing content delivery and enhancing performance. CDNs like Cloudflare, Akamai, and Netlify distribute assets globally, reducing latency. Proxy servers, such as Nginx and HAProxy, bestow features like caching, security, and anonymity onto the application's deployment environment.

Containerization technologies such as Docker and Podman, along with orchestration platforms like Kubernetes, have revolutionized deployment by encapsulating applications into portable containers. Kubernetes automates deployment, scaling, and management, offering features like self-healing and autoscaling. Docker Compose and Docker Swarm also provide less complex alternatives.

IaC (*Infrastructure-as-Code*) tools like Terraform, Ansible and SST[18] automate infrastructure provisioning, enabling efficient management through declarative code. Serverless computing, exemplified by AWS Lambda, abstracts

[18] https://sst.dev/

server management, allowing developers to focus on code. Finally, edge computing reduces latency by pushing computation closer to users.

Cloud platforms like AWS, GCP (*Google Cloud Platform*), and Azure offer comprehensive services for hosting and scaling web applications. Services like Amazon EKS and AWS Amplify simplify application management. Cloud computing provides scalability, reliability, cost-effectiveness, and continuous innovation.

Object storage, utilizing cloud architecture, is scalable and cost-effective, ideal for unstructured data like images and videos. MinIO, an open-source object storage server, is compatible with the very popular Amazon S3. Buckets within object storage organize data logically, finding applications in website hosting, data backup, big data analytics, and machine learning.

II.14 Software Factory

A Software Factory is a holistic environment facilitating the efficient development of software applications, encompassing various tools and practices that streamline the SDLC (*Software Development Lifecycle*).

Monolithic repository (shortened to Monorepo) tools, defined in Nrwl (2023), are an important choice to consider when managing large codebases and complex projects. Examples include Bazel, which supports multi-language monorepos, and Lerna, used for managing JavaScript projects. Nx, Turborepo, and Bit are other tools offering unique advantages in monorepo-style development, such as improved build times and component-based code sharing.

Other development tools like code formatters, such as Prettier, and linters, such as ESLint, contribute to code quality and consistency. Test runners like Jest for unit testing and frameworks like Selenium, Cypress, or Playwright for end-to-end testing ensure code reliability and functionality. Furthermore, proper use of Environment Variables enhances code portability and security by separating configuration settings from the codebase.

Content Management

Traditional CMS (*Content Management System*) platforms (e.g., WordPress), headless CMS (e.g., Contentful), and static site generators (e.g., Jekyll), offer diverse solutions for content management. CMS typically consist of two main components: a content management application (CMA), distributed to less technical content owners, and a content delivery application (CDA), for developers to retrieve organized data.

User Authentication

Security and Identity are critical aspects of web applications and APIs. RBAC (*Role-Based Access Control*), Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Attribute-Based Access Control (ABAC) are security models ensuring controlled access. Authentication providers like Auth0 and Keycloak, multifactor authentication, passwordless login, and self-hosted authorization platforms like Cerbos further strengthen security measures.

Notifications

Notification Providers like Pusher, Twilio, and OneSignal offer cross-platform push notification services, addressing challenges related to device compatibility, battery life, and user experience.

Push notifications present some challenges to developers such as delivering notifications promptly and reliably across devices, networks, and user preferences, securing user data and respecting privacy while personalizing notifications for optimal engagement and optimizing performance and monitoring analytics to ensure a seamless notification experience.

E-mail

SMTP (*Simple Mail Transfer Protocol*) and IMAP (*Internet Message Access Protocol*), the pillars of email communication, work in tandem to ensure seamless message delivery and organization within user inboxes. SMTP handles the efficient routing of outgoing emails, while IMAP diligently fetches and sorts incoming messages, keeping users' inboxes up-to-date and organized. Email servers like Postfix and Sendmail handle incoming and outgoing traffic, while tools like Mailhog provide a testing environment. Email service providers like SendGrid and Mailchimp simplify email marketing.

Gmail, with its user-friendly interface and integration with Google's ecosystem, has become the dominant email provider. Its market share continues to grow, raising concerns about a potential monopoly in the email space.

Documentation

Documentation tools like JSDoc, Swagger/OpenAPI (previously mentioned in Section II.4.5), Docusaurus, and Read the Docs automate the generation of comprehensive and up-to-date documentation for codebases. They usually rely on SSG for the effect.

Content Protection

DRM (*Digital Rights Management*) technologies, exemplified by Google's Widevine, protect digital content by encrypting and controlling access. Initiatives such as Defective by Design[19] argue against the use of DRMs, seeing them as a threat to innovation in media, the privacy of readers, and freedom for computer users. Obvious examples of DRM usage in popular applications include Spotify and Netflix. They encrypt the downloaded content before storing it on the user's device, ensuring that only authorized users can access and play it.

Maintenance

Monitoring and observability equip organizations with insights into the health, performance, and behavior of their software systems, enabling proactive issue identification, optimization, and resource allocation. Tools like OpenTelemetry, Prometheus, Grafana, and Axiom facilitate this process by collecting, analyzing, and visualizing data from various sources.

OpenTelemetry serves as a unified data collection platform, while Prometheus stores and analyzes metrics to provide a comprehensive overview of system health and performance. Grafana visualizes metrics and logs, enabling actionable insights into system behavior. Finally, Axiom identifies and troubleshoots distributed systems, pinpointing performance bottlenecks and other issues.

Analytics

Very recently, in the summer of 2023, concerns about data privacy and regulatory compliance have led to a growing demand for alternative analytics solutions to Google Analytics. Google Analytics is not GDPR-compliant (*General Data Protection Regulation*), as it transfers user data to servers in the United States, where it is processed under different privacy standards. This has led to several EU countries, including Sweden[20]), to ban the use of Google Analytics.

To address these concerns, open-source analytics libraries have emerged as viable alternatives. Matomo and Umami[21] are two popular open-source analytics platforms that are designed to respect user privacy and comply with GDPR. These libraries offer a variety of features, including website traffic tracking, user behavior analysis, and conversion funnel tracking. They also provide user-friendly interfaces for visualizing and analyzing data.

[19] https://www.defectivebydesign.org/
[20] https://www.imy.se/en/news/four-companies-must-stop-using-google-analytics/
[21] https://umami.is/

II.15 User Interactivity Design

UI and UX (*User Experience*) development is essential for crafting visually appealing and user-friendly digital products, encompassing both the look (UI) and overall UX.

Comprehensive UI design involves understanding user needs, defining goals, creating flows, designing information architecture, and continuous testing. Key design principles include visual hierarchy, contrast, balance, consistency, simplicity, and feedback. Design tokens ensure consistency within a system, and frameworks like Storybook aid in component testing.

Atomic Design, a UI creation methodology, structures interfaces from small, reusable components, fostering systematic and modular design. It progresses from Atoms (basic UI units) to Molecules, Organisms, Templates, and finally Pages, facilitating consistency and scalability.

II.15.1 Evolution of CSS

CSS (*Cascading Style Sheets*), fundamental in web development, styles HTML elements, governing visual aspects like colors and spacing. It uses rules to define element appearance, offering flexibility for creating responsive and visually appealing web interfaces.

CSS limitations, like lack of variables, can impact large-scale applications. Solutions include CSS preprocessors (Sass, LESS, PostCSS) and CSS-in-JS (styled-components, Emotion), enhancing maintainability and scalability.

Later, Tailwind CSS popularized the concept of atomic CSS, a utility-first framework, which streamlines styling with pre-defined classes. It provides rapid prototyping and consistent styling, making it advantageous for intricate designs. UnoCSS is a recent alternative, but Tailwind's popularity prevails.

Critical CSS techniques are used to improve page loading by prioritizing essential styles. Other techniques like minifying, lazy loading, and caching address these performance challenges.

Container Queries, a recent CSS feature, enable responsive design based on container dimensions, which marks a new approach instead of media queries (based on screen width) for responsive design.

II.15.2 UI Libraries

UI libraries offer a diverse range of options, catering to developers seeking different levels of flexibility, interactivity, theming capabilities and readiness out-of-the-box.

Primitive component libraries (e.g., React Aria[22]) serve as foundational building blocks, offering JavaScript code, usually in the form of hooks, for granular component creation and composability.

Unstyled (also called headless) component libraries such as Radix UI[23] are one step above, and specialize in providing structural elements without design-specific attributes, aligning with accessibility standards for inclusive UIs.

Tailwind CSS provides a concise and powerful way to style HTML elements, trading DRY (*Don't Repeat Yourself*) principles in favor of code colocation of structure its styles. By combining the Tailwind utility classes, developers can create custom and visually appealing UIs with greater flexibility and control.

Style systems, like Material Design and Fluent, offer fully styled components, ensuring consistency but potentially lacking distinctiveness. Their advantage is their capabilities for quick prototyping and enhancement of collaboration and maintainability within an organization. They're especially relevant when trying to mimic native styles.

An old favorite, the Bootstrap framework offers fully-styled components through CSS classes while frameworks like MUI React offer them through JSX.

"@shadcn/ui"[24] has gained a lot of traction recently for integrating Tailwind CSS styling with Radix UI behaviors, and offering the full code of those pre-built components through a CLI (*Command-Line Interface*) for easy integration into React projects where you own the code instead of having it abstracted away into the project's dependencies.

On a more opinionated step, NextUI[25], a React UI library, combines Tailwind CSS and React Aria, providing accessible and customizable components for highly personalized projects.

[22] https://react-spectrum.adobe.com/react-aria/
[23] https://www.radix-ui.com/
[24] https://ui.shadcn.com/
[25] https://nextui.org/

Motion libraries like Framer Motion and GSAP add the power of animations to apps, while tools like Class Variance Authority (CVA)[26] offer an opinionated approach to efficiently manage CSS class variants.

Icon libraries like Font Awesome and Iconify[27] have evolved to support tree shaking, which means they only bundle the necessary icons to the user.

A next-gen look at UI libraries is Meta's tentative StyleX[28], which looks to solve CSS by using an approach that more closely resembles what is done for native apps.

More recently, Tailwind CSS-based libraries like Flowbite and Preline UI[29] offer separated HTML with Tailwind classes, streamlining styling without specific React integration.

II.15.3 UI Trends

Current design trends encompass interactive onboarding, progress bars, impactful hero sections, Bento grids, and elements like progress bars and infinite scrolling. Figure 5 illustrates one of the most popular examples of bento grids, which served as inspiration for other Apple-related products.

[26] https://cva.style/docs
[27] https://iconify.design/
[28] https://stylexjs.com/
[29] https://preline.co/

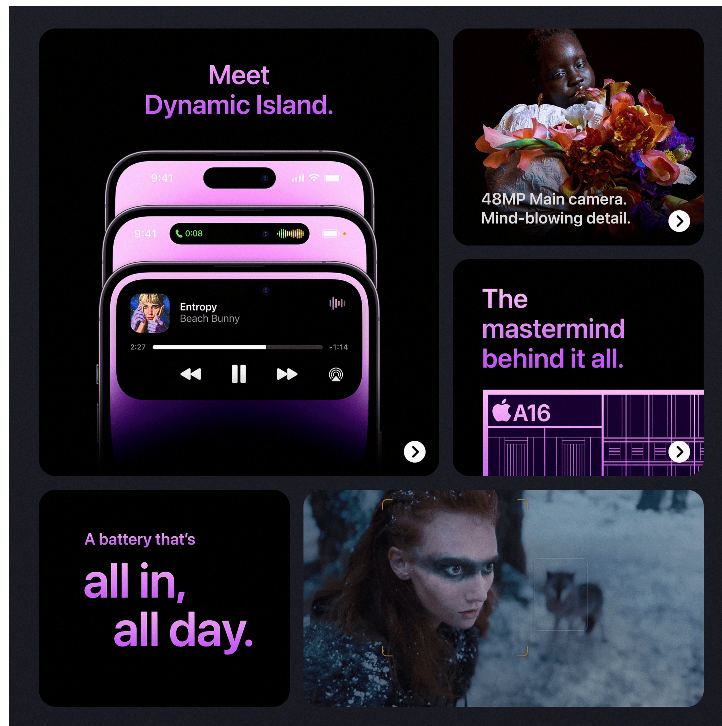


Figure 5: Landing page for presentation of the iPhone 14, displaying prominent use of Bento grids (Source: https://www.apple.com/iphone/)

For the later part of the 2010s the dominating style was minimalism and flat design. However, as of 2023, we're seeing wider adoption of neumorphism, which is a hybrid style that combines flat design with subtle skeuomorphism. It uses shadows and gradients to create a sense of depth and dimension, but it avoids using realistic textures.

II.16 Software Development Strategies

Software development methodologies have evolved over time to meet changing industry needs. The Waterfall methodology, the oldest and sequential, is still used for microprojects, but has clear drawbacks in rigidity and limited client involvement. The next step forward, Rational Unified Process (RUP), based on UML (*Unified Modeling Language*) and Object-Oriented Design, faces issues like complexity and excessive documentation. To contrast, Rapid Application

Development (RAD) emphasizes quick delivery but has little measures against scope creep.

Nowadays software companies mostly favor its evolution, Agile, focusing on iterative development, thought it faces challenges in predictability and documentation. Scrum, a popular Agile framework, involves short sprints and constant feedback. Kanban uses visual workflow management, focusing on a backlog with a column for each different stage of implementation for features (usually “To Do”, “In Progress” and “Completed”). Lean, originating from manufacturing, provides guidelines in optimizing efficiency, quality, and customer satisfaction.

A different approach is DevOps, which combines development and IT operations, enhancing software delivery speed, frequency, and quality. CI automates code integration and testing, while CD automates code deployment. CI/CD tools like Jenkins and GitHub Actions streamline these processes.

TDD (*Test-Driven Development*) prioritizes testing throughout development before following a cycle of writing tests, code, and refactoring. TDD requires discipline but ensures code reliability.

Various models suit different projects, considering factors like location, team size, budget, industry, and project type. In-house, nearshore, offshore, and Global Software Development (GSD) models each offer distinct advantages.

II.16.1 Organization of code

Source code is the fundamental building block of software development projects. Organizing and sharing source code, especially for projects that follow strict software development methodologies, requires tools that can back up and manage changes from all sources.

VCS (*Version Control Systems*) like Git, Mercurial, and Apache Subversion track code changes, enabling collaboration, rollback options, and code evolution monitoring. Git and Mercurial are distributed, allowing offline collaboration, with Git being highly efficient for large projects. Subversion is centralized, making collaboration more challenging but simplifying permission management.

Supported by platforms like GitLab, GitHub, and Bitbucket, Git facilitates global repository sharing, making it by far the preferred choice for collabora-

tive projects. Some newer solutions like Graphite[30] attempt to streamline the GitHub experience further.

Packages/modules group related code, with modules usually referring to file-based grouping and packages to directory-based. While enhancing code modularity and reusability, coordinating them in pipelines and synchronizing releases can be challenging. Packages are generally ideal for organizing libraries and dependencies in most projects.

Standalone repositories manage single projects/components, suitable for small projects or those outside larger codebases. Simple to create and manage, standalone repositories are portable and can be hosted on any version control-supporting service. However, integration challenges and the risk of duplicate code inconsistencies may arise.

On the other hand, monorepos store all company/department code in a single repository, simplifying code sharing but requiring careful maintenance, especially for large projects. Integrated codebases in monorepos contain all project code in a single unit, while package-based codebases divide code into smaller, independent packages. Package-based codebases are more maintainable and updatable but can be complex to set up and deploy.

II.16.2 Project Management Tools

Tools like Jira, Confluence, Notion, and GitHub Projects aid project planning, tracking, and management. Jira focuses on issue and project tracking, Confluence facilitates collaboration and documentation, Notion combines note-taking and project management, and GitHub Projects integrates with code repositories.

II.16.3 Software Diagrams

UML provides a standard for modeling software systems. Tools like StarUML, PlantUML, Draw.io, MermaidJS, and Excalidraw help create UML diagrams with varying levels of detail and ease of use.

II.16.4 Deployment Strategies

Deployment strategies, such as Blue-Green, Canary, and Rolling deployments, ensure efficient and safe introduction of new code or features to production environments. Blue-Green involves running parallel environments, Canary re-

[30] https://graphite.dev/

leases to a small subset before expanding, and Rolling deploys incrementally to subsets while allowing easy rollback.

II.17 E-commerce

In the ever-evolving landscape of e-commerce, the role of software developers is crucial in adopting to an efficient business model, that engages customers. This section delves into essential considerations from a software development perspective.

E-commerce's advantage lies in its access to abundant data, necessitating robust tools for analysis. Integration of CRM (*Customer Relationship Management*) platforms like Salesforce Commerce Cloud is vital. These platforms utilize customer data to deliver targeted messages for repeat sales. Load time optimization is equally critical, as a 3-second load time can prompt user abandonment (Shellhammer and Neel (2016)). To enhance customer experience, strategies like referral programs, coupon management systems, chatbots, and CRM systems are employed. Social media integration fosters consumer engagement, while optimizing logistics, inventory management, and offering sustainable fulfillment solutions ensure a seamless customer journey. A hassle-free returns process through well-designed interfaces also contributes to customer loyalty.

Choosing the right e-commerce framework is pivotal if not making a solution from scratch. Shopify is renowned for ease of use but has customization limitations. Adobe Commerce offers extensive customization for businesses seeking control, but is complex to set up. WooCommerce, deeply integrated with WordPress, offers unparalleled flexibility and familiarity for WordPress users, but its reliance on the PHP ecosystem limits its reach and flexibility. In contrast, MedusaJS, an open-source headless e-commerce framework, stands out with its modularity and extensibility, making it a compelling choice for those seeking a more flexible and scalable e-commerce solution.

A full e-commerce system comprises the commerce backend, admin panel, and frontend. The backend manages authentication, cart, customer interactions, pricing, and more, while the admin panel controls product, order, and customer management. The frontend, or storefront, is where customers interact, view products, and conduct transactions.

E-commerce platforms rely on various integrations. Payment providers like PayPal and Stripe process payments securely, fulfillment providers like ShipStation manage logistics and analytics tools like Segment track user behavior.

Robust search functionality is crucial, with solutions like Elasticsearch and Meilisearch. CMS integration enables easy management, with Strapi being a popular choice. Notification tools like Twilio manage order updates.

II.18 Social Networking

Since their introduction, social networks have evolved from monolithic applications with traditional databases and simple caching to distributed applications with NoSQL databases and distributed caching, all while incorporating robust security measures to support a growing number of users and interactions.

Social media platforms use a mix of human and algorithmic moderation. Automated systems review content quickly, but human moderators are essential for accuracy. Platforms work with fact-checkers, implement policies, and develop algorithms to address misinformation and hate speech, striving to balance free speech protection.

II.19 Recommender Systems

For deciding which content to show to users, recommender systems are powerful tools that employ a variety of machine learning algorithms to generate recommendations. These algorithms can be categorized into four main types: collaborative filtering (CF), content-based filtering (CBF), hybrid recommender systems, and deep learning.

Early recommender systems relied on hand-crafted rules, which were often based on intuition or expert knowledge. While effective for simple tasks, they lacked adaptability to changing user behavior or data, and scalability to large datasets.

As machine learning algorithms matured, recommender systems incorporated these techniques into their decision-making process. Machine learning algorithms can learn from data and identify patterns, leading to improved accuracy and personalization.

The latest generation of recommender systems, powered by deep learning, utilizes artificial neural networks to learn complex patterns in data, enabling even more accurate and personalized recommendations.

II.20 Book Reader Software

Book reader software has evolved from early e-readers with black-and-white screens to dedicated e-readers with features like wireless downloads. Multipurpose devices like tablets dominate the market now. Notable examples include Calibre, Adobe Digital Editions, Kindle, Apple Books, and Google Play Books.

For e-book file formats, PDFs preserve formatting but can be challenging on small screens. EPUB is open and reflowable, suitable for various devices. MOBI and AZW3 are Amazon's proprietary formats. Well-formatted text is essential for a good reading experience.

E-book files require compact size, compatibility, reflowable text, internal links, multimedia support, and DRM handling for protected content.

II.21 Audiobook Player Software

Audiobook player software decodes and plays audio files. Their evolution spans from CD players to versatile smartphone apps. Future trends may include AR/VR integration and enhanced smart speaker compatibility.

MP3, M4A, and FLAC are common audiobook formats. Audiobook files need chapters, bookmarks, and playback position tracking.

Variable playback speed, sleep timers, and cross-device integration enhance user experiences.

Notable audiobook players include Smart AudioBook Player, Audible, Google Play Audiobooks, Apple Audiobooks, and OverDrive.

Playback speed adjustment, bookmarking, and sleep timers are common features, implemented differently by various apps.

Understanding the intricacies of these software domains empowers users to navigate and utilize these tools effectively.

II.22 Similar Products

In this section, we perform a comparative analysis of various domains within the *SHIDA* application, juxtaposing them with similar benchmark products available in the market. This analysis aims to uncover the strengths, weaknesses, and unique features that distinguish *SHIDA*'s services from the following benchmark products.

II.22.1 What is a super-app?

A super-app is a mobile application that extends its services well beyond its core functionality. Often dubbed as all-in-one apps, they are versatile tools capable of handling a multitude of tasks, ranging from messaging and social networking to e-commerce and transportation.

Asia, in particular, has embraced super-apps with open arms, where they have seamlessly integrated into the daily routines of countless individuals. Take, for instance, the Chinese super-app WeChat, boasting over 1.3 billion active users[31] and offering an extensive array of services, including messaging, social media, payments, and e-commerce.

The allure of super-apps lies in their ability to streamline the user experience by providing access to a plethora of services within a single application. This proves invaluable, especially in bustling urban settings where individuals may require on-the-go access to diverse services. super-apps are highly customizable to cater to each user's unique preferences. Users can cherry-pick the services they desire and tailor how they interact with them, rendering super-apps exceptionally user-friendly and engaging.

Super-apps foster an ecosystem of tightly integrated services, simplifying the process for users to discover and utilize the services they require.

Businesses stand to benefit significantly from super-apps as they offer a gateway to a vast and engaged user base. The sheer magnitude of users and the diverse range of services provided by super-apps enable businesses to collect extensive user data. This data can be harnessed to enhance the user experience and conceive innovative products and services. For example, a super-app might leverage user travel data to develop a novel transportation service.

[31] https://www.statista.com/statistics/255778/number-of-active-wechat-messenger-accounts/

Super-apps can be monetized through various channels, including advertising, in-app purchases, and commissions, opening up new and potentially lucrative revenue streams for businesses.

II.22.2 WeChat (General)

WeChat is a super-app that offers messaging, social networking, mobile payments, and other services, dominating the Chinese market.

SHIDA offers a comprehensive social network platform that integrates various services, including book clubs, matchmaking, e-learning, and more, providing users with a holistic experience.

Both *SHIDA* and WeChat focus on providing multiple services within a single app. While WeChat's strength lies in its dominance in the Chinese market and wide-ranging services, *SHIDA* aims to offer specialized services for literature-centered social networking focused on a gap in the Horn of Africa market, making it unique in its approach.

II.22.3 Kindle & Audible (Digital Library)

SHIDA's Library platform offers a diverse range of literature, including eBooks and audiobooks, available for purchase or subscription, catering to the reading preferences of its users.

In comparison, benchmark products like Amazon Kindle and Audible also provide digital content consumption, with Amazon Kindle offering eBooks and Audible focusing on audiobooks. However, it's worth noting that *SHIDA*'s Library provides users with a more comprehensive reading experience in a single platform by merging all formats of a book together into a single product.

Moreover, *SHIDA* has ambitious plans to develop specialized hardware that seamlessly integrates with the app, allowing users to read books in a user-friendly and dedicated manner. This initiative is akin to how the Kindle tablet works, enhancing the overall reading experience and aligning with *SHIDA*'s commitment to providing innovative and user-centric services.

II.22.4 Clubhouse, Twitter Spaces & Discord (Talk Rooms)

SHIDA's Talk Rooms service is similar to Twitter Spaces and Clubhouse, offering talk rooms and chat lobbies for group discussions and interactions among users.

Twitter Spaces, which was inspired by Clubhouse, is well-known for their real-time audio-based social networking, allowing users to participate in discussions and listen to live conversations.

The two platforms mentioned above and Discord offer different approaches to audio chat rooms, with a notable dynamic focus on events, while Discord focuses on communities and structure.

In Twitter Spaces, users can spontaneously create live audio chat rooms within the context of their account. These Spaces are temporary and do not persist beyond the duration of the audio chat session, emphasizing a more dynamic and real-time event-based interaction.

On the other hand, Discord is a separate platform that revolves around premade servers and channels, designed for ongoing and structured discussions within dedicated communities. The focus on communities in Discord provides a more persistent and organized environment for users to engage in ongoing conversations and interactions.

SHIDA's Talk Rooms strike a balance between the dynamics of Twitter Spaces and the structured communities of Discord. Similar to Twitter Spaces, *SHIDA*'s Talk Rooms allow users to create live audio chat rooms for real-time interactions and discussions, emphasizing the event-driven aspect. However, unlike Twitter Spaces, *SHIDA*'s Talk Rooms offer more persistence by allowing users to store and access recorded audio sessions after the event, providing a blend of both dynamic and lasting content.

In contrast to Discord, where pre-made servers and channels are the foundation, *SHIDA*'s Talk Rooms offer a more flexible approach. While *SHIDA*'s platform does have user-generated "Clubs" for organizing discussions about specific interests, the Talk Rooms within these Clubs remain temporary, aligning with the event-based focus. This approach gives users the freedom to participate in ad-hoc discussions while still being part of thematic communities.

Overall, *SHIDA*'s Talk Rooms provide a dynamic and event-driven audio chat experience while offering the option for more lasting content through

recordings. They sit in between the spontaneity of Twitter Spaces and the structured communities of Discord, providing users with a versatile platform for engaging in live audio conversations within both temporary and thematic settings.

II.22.5 Meetup (Matchmaking)

SHIDA Book Clubs is a literature-centered ecosystem that offers a diverse range of literature, including e-books and audiobooks, creating a vibrant reading community. It provides unique features for writers, such as privileged connections to publishing companies and partners, to help aspiring authors gain exposure and potential publishing opportunities. In contrast, Meetup is a general event-centric platform that includes literary events and book clubs among various interests but lacks specialized features for writers. *SHIDA* Book Clubs focus on literature and writer support, while Meetup serves as a broader community engagement platform.

II.22.6 Medium (Knowledge Hub)

Medium, a renowned platform, serves as a comprehensive Knowledge Hub. It provides a space for writers and experts to share articles, essays, and insights on a multitude of topics.

SHIDA, with its Library platform, offers a different but equally engaging approach. While Medium emphasizes user-generated content and the sharing of knowledge through articles, *SHIDA* focuses on curated literature, including eBooks and audiobooks. *SHIDA*'s commitment to literature and its supportive ecosystem for writers distinguishes it from Medium.

II.22.7 Masterclass (Knowledge Hub)

Masterclass is a prominent platform in the Knowledge Hub domain, offering online courses and tutorials taught by renowned experts in various fields.

SHIDA stands out by providing access to an expansive range of literature, including eBooks and audiobooks, fostering an intellectual community. While Masterclass offers learning through courses, *SHIDA* enriches users' knowledge through literature and promotes literature-centered discussions and interactions.

II.22.8 Goodreads (Digital Library)

Goodreads is a widely recognized Digital Library, primarily focused on book recommendations, reviews, and discussions.

SHIDA's Library platform takes a distinctive approach, offering both eBooks and audiobooks, coupled with social networking features such as book clubs and matchmaking. While Goodreads excels in book discovery and discussions, *SHIDA* extends the digital library experience by providing diverse content and immersive engagement options.

II.22.9 Tinder/Bumble (Matchmaking)

Tinder and Bumble are prominent players in the Matchmaking domain, connecting individuals based on their interests and preferences.

SHIDA incorporates matchmaking features within its comprehensive social network platform. However, *SHIDA*'s matchmaking services are tailored to less carnal, literature-centered interactions, creating meaningful connections among users who share a passion for books and literary discussions.

II.22.10 Novellic (Book Clubs)

Novellic specializes in Book Clubs, offering a platform for readers to join or create book clubs and engage in literary discussions.

SHIDA Book Clubs also focus on fostering a vibrant reading community, with a diverse range of literature, including eBooks and audiobooks. *SHIDA* provides additional support for writers and offers a more extensive literary ecosystem, differentiating it from platforms like Novellic.

II.22.11 itch.io (Graphics/Games)

itch.io is a well-established platform within the game development ecosystem, primarily designed to support indie game developers and hobbyists in sharing and selling their game creations. *SHIDA* aims to draw inspiration from itch.io's success and introduce a lighter version of its marketplace services. This new feature will enable users to buy and sell games in a user-friendly and supportive environment, catering to the needs of hobbyist game creators within the *SHIDA* community.

III Methodology

The *SHIDA* project was implemented using a holistic approach, which considered all aspects of the project, including the people, processes, and technology involved. This approach sought to result in improved project success, reduced risk, increased efficiency, and improved stakeholder satisfaction.

III.1 Project Requirements

Requirement analysis is a crucial step in software development, involving the gathering, analysis, and documentation of system requirements. The primary objective is to ensure that the software system aligns with user and stakeholder needs.

This analysis typically takes place early in the development process, following the creation of the initial project plan and project scope definition.

III.1.1 General, High-Priority Requirements

1. Agile and Automated Development

   Description: Utilize a DevOps CI/CD pipeline for automated development and deployment. This approach ensures rapid and reliable platform development and release.

2. Scalability to Social Network Size

   Description: Implement the SAFe (*Scaled Agile Framework*) and the LeSS (*Large-Scale Scrum*) framework to adaptively scale platform development and maintenance.

3. High Control Over Technology

   Description: Embrace free and open-source software to prevent vendor lock-in. While this approach grants greater control, it may require additional effort for platform maintenance and upgrades.

4. Accessibility for African Conditions

   Description: Design the platform to cater to users in African conditions, characterized by low bandwidth and modest-performance devices. This entails enabling offline access and global distribution through a CDN.

5. Mobile-First, Cross-Platform Design

   Description: Ensure platform accessibility across all devices, irrespective of screen size, performance, or connectivity. Implement techniques like graceful degradation and progressive enhancement:

   - Minimize client-side JavaScript usage.
   - Employ responsive design principles.
   - Apply progressive enhancement to enhance features for capable devices.

- Thoroughly test the platform on various devices to ensure universal accessibility and usability.

6. System Modularity

   Description: Architect the system in a modular fashion, comprising independent modules developed, tested, and deployed separately. Establish well-defined interfaces for communication between modules.

7. Source Code Protection

   Description: Implement stringent source code protection measures to prevent unauthorized access, modification, and disclosure:

   - Store source code securely in a private Git repository hosted on platforms like GitHub or GitLab.
   - Enforce strict access controls using two-factor authentication and RBAC.
   - Implement code review and auditing practices to identify and rectify security vulnerabilities.
   - Employ a WAF to shield against common web attacks.
   - Keep the platform's software updated with the latest security patches.

8. GDPR Compliance

   Description: Ensure platform compliance with relevant data protection regulations, including the GDPR. This necessitates lawful, transparent, and user-centric handling of personal data, granting users control over their data.

9. User-Friendly Design

   Description: Prioritize a user-centric design, focusing on a seamless UX suitable for users of varying technical backgrounds:

   - Employ a clear and concise design with easy navigation.
   - Maintain consistent design elements and patterns.
   - Use plain language for clear communication.
   - Provide user-friendly feedback.
   - Ensure accessibility for users with disabilities.
   - Optimize task completion speed and user satisfaction.

III.1.2 Per Service

Section I.2.3 describes the product structure for the project vision phase. Requirements analysis aims to go one step further by detailing how the system's components should behave and what constraints they should follow.

1. Digital Library

   Functional:

   - Enable browsing and searching of the digital library by various criteria.
   - Facilitate viewing and downloading of e-books and audiobooks.
   - Support user management of digital library collections.
   - Offer a monthly subscription for temporary access to featured books.

   Non-functional:

   - Ensure universal accessibility across all devices.
   - Implement robust security measures to protect user data.
   - Safeguard digital library products against piracy.

2. Reader Clubs

   Functional:

   - Enable user participation in and creation of reader clubs.
   - Facilitate book discussions among club members.
   - Support organization and participation in platform-organized club events.
   - Provide text-based discussion and integration with Talk Rooms.

   Non-functional:

   - Ensure scalability to accommodate future growth in terms of clubs, members, features, and functionality.

3. Writer Clubs

   Functional:

   - Enable user participation in and creation of writer clubs.
   - Facilitate sharing of writing and feedback among club members.
   - Offer support for publishing books on the platform.

Non-functional:

- Protect user data, including the privacy of user writing.
- Implement manual user reviews before club participation.

4. Talk Rooms

   Functional:

   - Allow creation and participation in talk rooms for various topics.
   - Enable real-time voice chat in talk rooms.
   - Provide moderation and access management for talk rooms.

   Non-functional:

   - Ensure user-friendly and efficient usability.
   - Support a large number of concurrent users.
   - Maintain low latency (below 100 milliseconds).

5. Storefront

   Functional:

   - Allow users to purchase books for their digital libraries.
   - Enable ordering of physical books and merchandise.
   - Facilitate product browsing and searching.
   - Maintain shopping cart persistence during navigation.
   - Allow users to leave reviews and ratings for products.
   - Notify users of new releases through notifications or interface features.

   Non-functional:

   - Ensure secure handling of user data, including personal and payment information.
   - Support a large number of concurrent users.
   - Ensure platform availability with minimal downtime (at least 99.9% uptime).

6. Knowledge Hub

   Functional:

- Enable browsing and searching of the knowledge hub by category or recency.
- Allow user contributions of knowledge through article creation and editing.
- Support inclusion of images and videos in articles.
- Facilitate article discussions via text-based discussion or integration with Talk Rooms.
- Offer a monthly subscription for access to premium articles.

Non-functional:

- Strive for high accuracy in knowledge content.
- Present information objectively, without bias, and encompassing all sides of a topic.
- Implement manual reviews for Knowledge Hub creators.

7. Villages Wiki

   Functional:

- Enable browsing and searching of stories by village, country, and other criteria.
- Allow users to submit their chronicles.
- Accept translations for existing chronicles.

Non-functional:

- Implement manual review for newly submitted stories before publication.

8. Matchmaking

   Functional:

- Offer a monthly subscription service.
- Enable user profile creation and viewing.
- Support user searches and matches based on preferences.
- Facilitate user communication through text chat.

Non-functional:

- Ensure accurate matchmaking based on user preferences.

- Respect user privacy preferences.

9. Charity Hub

   Functional:

   - Provide detailed charity information, including mission, programs, and financial statements.
   - Enable user donations to charities through various payment methods.
   - Allow users to track donations and view donation history.

   Non-functional:

   - Maintain transparency in donation usage and platform operation.

10. Creativity Hub

    Functional:

    - Allow users to share creative works, including art, animation, photography, and games.
    - Enable user collaboration on creative projects.
    - Facilitate discovery and enjoyment of other users' creative works.

    Non-functional:

    - Implement manual user reviews before participation in the Creativity Hub.

11. Money Transactions

    Functional:

    - Enable users to send and receive money with competitive rates.

    Non-functional:

    - Ensure secure handling of user data.
    - Comply with all applicable financial regulations.

12. Travel Packages

    Functional:

    - Allow browsing and searching for travel packages using various criteria.
    - Provide detailed information about each travel package, including itinerary, price, and inclusions.

- Enable booking of travel packages through various payment methods.
- Allow users and providers to manage bookings.

Non-functional:

- Ensure reliable real-time booking with support for a large number of concurrent bookings.

Some requirements are more detailed than others, but all of them are essential to the success of the project. The further detailing of the requirements has been tasked to the team.

III.2 Software Development Process

Since the software development process was largely dictated by the stakeholder's requirements, a comprehensive analysis of the most suitable approach was mostly overlooked. Nevertheless, the chosen methodologies are considered state-of-the-art and typically perform well. They may, however, face challenges when implemented in specific scenarios.

III.2.1 Scaled Agile Framework (SAFe)

SAFe (Scaled Agile Framework) is an organizational and workflow pattern for implementing agile practices at an enterprise scale (as explained in Piikkila (2020)). It promotes alignment, collaboration, and delivery across agile teams, incorporating agile software development, lean product development, and systems thinking.

SAFe emphasizes core values such as alignment, built-in quality, transparency, program execution, and lean-agile leadership. Its principles include taking an economic view, applying systems thinking, assuming variability, building incrementally, basing milestones on working systems, visualizing and limiting work in process, applying cadence, unlocking intrinsic motivation, and decentralizing decision-making.

SAFe provides a roadmap for implementation, with steps including training, creating a center of excellence, identifying value streams, launching agile release trains, and extending the portfolio. A diagram for the simpler version of this roadmap is included in Figure 6.

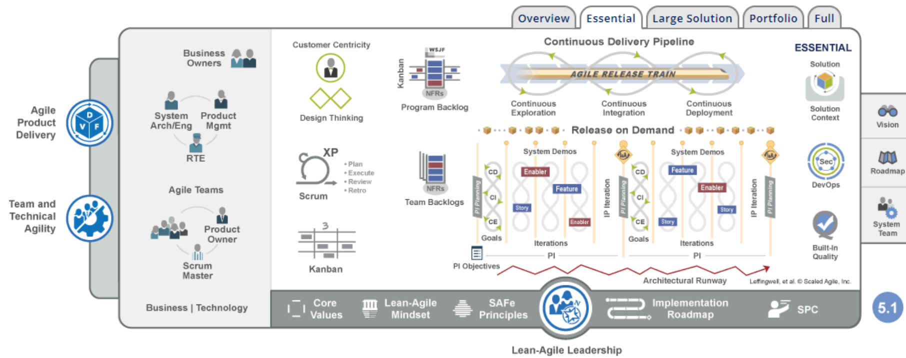


Figure 6: "Essential" configuration of SAFe 5 (Source: https://v5.scaledagileframework.com/)

Product Increment Planning

PI (*Product Increment*) Planning, is a large-scale planning event used in the SAFe. It is a two-day event that brings together all the teams and stakeholders involved in a program to plan the next PI.

The goal of PI Planning is to align all the teams on the program's goals and objectives, and to create a plan for delivering a working product increment at the end of the PI. The PI Planning event also serves as an opportunity for teams to identify and mitigate any risks or dependencies.

III.2.2 Large-Scale Scrum: More with LeSS

LeSS (Large-Scale Scrum) is an organizational system that maximizes adaptiveness by optimizing change and delivering value to customers and end-users. It takes a systems approach, recognizing that organizational adaptiveness requires impact and alignment across people, customers, structure, policies, processes, and practices.

LeSS promotes simpler structures to overcome barriers like complexity and single specialization, avoiding excessive roles, processes, and artifacts. It emphasizes continuous improvement, fostering a sense of purpose and ownership among teams. By following LeSS principles and encouraging experimentation, organizations can achieve systemic adaptiveness and drive continuous improvement.

III.2.3 DevSecOps and CI/CD

DevSecOps, short for Development, Security and Operations, is an approach that combines software development practices, security considerations, and IT operations into a collaborative and streamlined process (as explained in Red Hat (2023)). It aims to bridge the gap between these teams to deliver software faster, more securely, and with improved quality.

Key elements of DevSecOps include automation, continuous integration, continuous delivery, and continuous deployment (CI/CD). By automating tasks and integrating them into the development pipeline, DevSecOps helps reduce manual errors, accelerate software delivery, and ensure consistency.

Security is a fundamental aspect of DevSecOps, emphasizing the integration of security practices and considerations throughout the development process. It involves proactive identification and mitigation of vulnerabilities and threats, incorporating security controls, and performing regular security assessments.

DevSecOps also encourages a shift-left approach, where security is integrated early in the development cycle rather than being an afterthought. Security testing, code analysis, and vulnerability scanning are performed continuously to identify and address security issues early on.

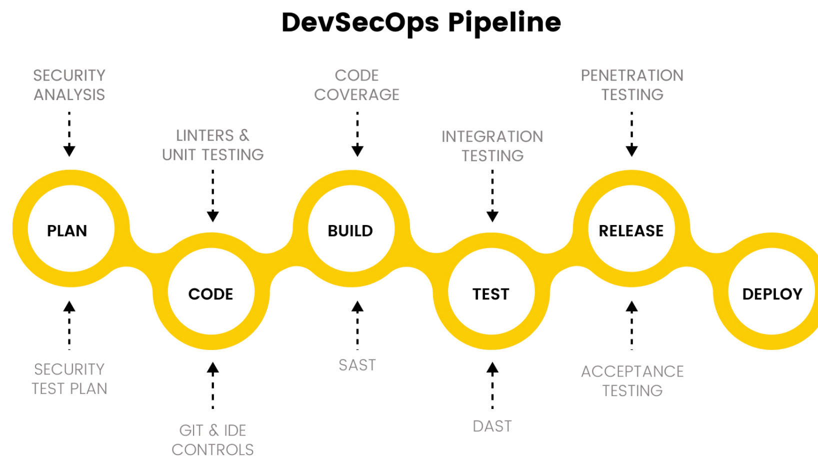


Figure 7: DevSecOps pipeline (Source: https://www.qentelli.com/thought-leadership/insights/devsecops-pipeline-factors)

III.3 Challenges

Developing software can be a challenging endeavor, especially when striving to create something entirely new. In the world of software development, one common challenge that many individuals and teams face is the tendency to procrastinate or overcomplicate the process. As aptly noted by McRill (2023), anxiety about creating new code often leads to a form of procrastination known as scope creep, future thinking, or premature optimizations.

This article sheds light on the various types of procrastination that can hinder software development progress and provides valuable insights into overcoming these challenges. It emphasizes the importance of adhering to well-defined requirements and the benefits of focusing on the "definition of done." Additionally, the article advocates for shipping small, frequent updates and embracing the iterative nature of software development.

In the sections that follow, a deep dive is done into some of these software development challenges, exploring effective strategies for addressing them, drawing inspiration from the wisdom shared in this article.

III.3.1 Over-Engineering

Over-engineering is the process of designing or building a system or product more complex or sophisticated than necessary. It can lead to wasted time, resources, and money, and it can make the system or product more difficult to maintain and update. In the *SHIDA* case, there's a very high risk of over-engineering, since the team is building a complex system with many features.

There are several strategies that the team could employ to prevent and reduce over-engineering. Some key considerations done throughout the project include:

1. **Starting with Clear and Concise Requirements:** The project started with a search for specific and unambiguous requirements.
2. **Using a Simple Design Approach:** Throughout the project, focus was put more and more on proven and well-understood technologies.
3. **Gathering Early and Frequent User and Stakeholder Feedback:** The product owner ensured the team that they were building something

needed and aligned with user requirements by presenting its concept to the public and looking for stakeholders.

4. **Prototyping and Test Before Deployment:** Potential issues and opportunities for simplification were found through prototyping.
5. **Monitoring Solution Performance:** Every week, a recap meeting was conducted to assess the solution's performance and justify added complexity.
6. **Focusing on the Value Proposition:** Unnecessary features were avoided through a focus and consistent refocus on the Minimum Viable Product (MVP).
7. **Using a Risk-Based Approach:** The platform isn't complete without books to buy and read, therefore the focus was put first on the digital library as a high-impact, likely risk.
8. **Adopting a Modular Design Approach:** The system was planned from the beginning to feature microservices and a vertical split approach to enhance its maintainability and flexibility.
9. **Leveraging Off-the-Shelf Components:** An extensive study was done to find out any existing components for the solutions sought, to save time, costs, and reduce complexity.
10. **Willingness to Say No:** Some non-essential features or requirements were dropped along the way to reduce complexity. One example of this was the decision to not implement a dedicated mobile application.

By following these strategies, the team attempted to prevent and reduce over-engineering, delivering a complex system while aligning with user and stakeholder needs.

III.3.2 Chesterton's Fence

Chesterton's Fence, introduced in Parrish (2021), is a principle that states that one should not remove a fence until one knows why it was put up in the first place. This principle is based on the idea that fences are not built arbitrarily, but rather for a specific reason. If one does not understand the reason for the fence, one may end up doing more harm than good by removing it.

The principle of Chesterton's Fence can be applied to many different areas of life, including business, government, and personal relationships. In the context

of software development, it can be used to guide decisions about whether to adopt or abandon certain technologies or practices.

Another example of Chesterton's Fence in the real world that applies to software development is the case of hierarchy-free teams. Some people believe that hierarchies are outdated and inefficient, and that companies should be run without them. However, hierarchies exist for a reason. They provide a clear structure for decision-making and communication. Without a hierarchy, it can be difficult to make decisions quickly and effectively.

Another example of Chesterton's Fence is the case of bad habits. Bad habits often develop to serve an unfulfilled need. For example, someone may smoke to relieve stress or to feel connected to others. If one tries to break a bad habit without addressing the underlying need, one is likely to fail.

The principle of Chesterton's Fence is a valuable reminder that one should not take things for granted. One should always question the *status quo* and seek to understand the reasons for things before making any changes.

III.3.3 Analysis Paralysis

Analysis Paralysis is a state of overthinking or overanalyzing a situation, which can lead to indecision or inaction. It is a common problem in software development, where there are often many possible solutions to a given problem. The *SHIDA* project team has been careful to avoid analysis paralysis by following a few simple guidelines:

- **Focus on the problem:** The team has focused on the problem at hand rather than trying to find the perfect solution. This has helped to avoid analysis paralysis and to keep the project moving forward.
- **Keep it simple:** The team has tried to keep things as simple as possible. This has helped to avoid analysis paralysis and to keep the project moving forward.
- **Don't overthink it:** The team has tried to avoid overthinking things. This has helped to avoid analysis paralysis and to keep the project moving forward.
- **Don't be afraid to make mistakes:** The team has tried to avoid being afraid of making mistakes. This has helped to avoid analysis paralysis and to keep the project moving forward.

- **Don't be afraid to ask for help:** The team has tried to avoid being afraid of asking for help. This has helped to avoid analysis paralysis and to keep the project moving forward.

III.3.4 Scope Creep

Scope Creep is the tendency for a project's scope to expand over time. It can be caused by a number of factors, including:

- **Poor requirements gathering and definition:** If the project's requirements are not clearly defined and documented, it can be difficult to determine what is in scope and what is out of scope.
- **Poor communication and collaboration:** If the project team does not communicate and collaborate effectively, it can be difficult to keep everyone aligned on the project's scope.
- **Poor change management:** If the project team does not have a robust change management process in place, it can be easy for changes to be made to the project's scope without proper approval.
- **Unrealistic expectations:** If the project's stakeholders have unrealistic expectations about what can be accomplished within a given timeframe and budget, it can lead to scope creep.
- **Feature creep:** Feature creep is a specific type of scope creep that occurs when new features are added to the project's scope without considering the impact on the project's timeline, budget, and quality.

To combat scope creep on the *SHIDA* project, the team has taken a number of measures, such as defining clear requirements, communicating and collaborating effectively, and managing changes carefully. These measures have helped to keep the project on track and to avoid scope creep.

III.3.5 Complexity Budget

SHIDA's methodology emphasizes the importance of simplicity and minimalism in software development. This is in line with the concept of a complexity budget (Gross (2020)), which is the idea that every application has a limited amount of complexity that it can handle. Once the complexity budget is exceeded, the application becomes difficult to understand, maintain, and extend.

There are a number of things that developers can do to manage their complexity budget, such as:

- Clearly defining the requirements for the application and prioritizing features based on their importance.
- Choosing the right tools and technologies for the job.
- Factoring the application into manageable components.
- Writing simple and readable code.
- Testing the application thoroughly.
- By following these guidelines, developers can help to ensure that their applications are reliable, maintainable, and scalable.

In the context of *SHIDA*'s methodology, the complexity budget can be used to make decisions about which features to include in the application and how to implement them. For example, if a feature is complex and not essential to the core functionality of the application, such as the matchmaking service, it may be better to defer its implementation until later.

The complexity budget can also be used to guide refactoring efforts. If a section of code is complex and difficult to understand, it may be worth refactoring it to make it simpler. However, it is important to be careful not to over-engineer the code, as this can add unnecessary complexity.

Overall, the complexity budget is a valuable tool that can help developers to build simpler and more reliable software applications.

III.3.6 Technological Debt

Technological Debt is a term used to describe the cost of maintaining and updating software systems. It is often used to refer to the cost of fixing bugs, adding new features, and making other changes to a software system.

Throughout the *SHIDA* project, the team has been careful to avoid accumulating technological debt. This has been done by following a number of best practices, such as:

- **Writing clean and readable code:** The team has strived to write clean and readable code. This has helped to reduce the amount of time spent on maintenance and updates.
- **Picking the right tools and technologies:** The team has sought to pick the right tools and technologies for the job, to avoid having to switch to a different technology later on.

- **Testing the application thoroughly:** The team should test the application thoroughly, to ensure that it is working as expected.

III.3.7 Future Thinking

Future Thinking is a form of procrastination that involves thinking about the future instead of taking action in the present. It can be caused by a number of factors, including fear of failure, perfectionism, and/or lack of confidence. Future Thinking can lead to missed opportunities and wasted time and resources.

To combat future thinking, the *SHIDA* project team has taken a number of measures, such as:

- **Focusing on the present:** The team has focused on the present rather than worrying about the future. This has helped to avoid future thinking and to keep the project moving forward.
- **Taking action:** The team has taken action rather than waiting for the perfect solution. This has helped to avoid future thinking and to keep the project moving forward.
- **Being open to feedback:** The team has been open to feedback from stakeholders and users. This has helped to avoid future thinking and to keep the project moving forward.
- **Being willing to make mistakes:** The team has been willing to make mistakes. This has helped to avoid future thinking and to keep the project moving forward.

III.3.8 Premature Optimizations

Premature Optimization is the act of optimizing a program or system before it is necessary to do so. Similarly to future thinking, it can lead to wasted time and resources, and it can also make the program or system more difficult to maintain and update. Premature optimization is a common problem in software development, where there are often many possible optimizations that can be made. Factors that can lead to premature optimization include lack of experience, performance anxiety, and inaccurate benchmarking.

The *SHIDA* project team, which is composed of mostly inexperienced developers, has been careful to avoid premature optimization by following a few simple guidelines. These include not optimizing until there is a task specifically assigned to it, avoiding over-engineering as detailed in Section III.3.1 and focusing on the definition of done as detailed in Section III.5.7.

III.4 Comparative Analysis of Application Frameworks

To make informed technology choices, we thoroughly compare various application frameworks, focusing on their appropriateness for the unique requirements of our project. We also delve into architectural paradigms and patterns that align with our project goals. Our aim is to select a framework that aligns with current technology trends and sets our project up for success. This comparative analysis provides a solid foundation for further exploration in subsequent sections.

III.4.1 Framework Evaluation Criteria

In assessing the suitability of application frameworks for the development of comprehensive web applications like the *SHIDA* project, a set of rigorous evaluation criteria becomes essential. These criteria serve as a compass, guiding the selection process and ensuring alignment with project objectives. Below, we outline the fundamental framework evaluation criteria:

1. **Scalability and Performance:** The ability of the framework to scale horizontally and vertically to accommodate a potentially large user base is paramount. Performance considerations, such as response times and resource utilization, directly impact the user experience.
2. **Flexibility and Extensibility:** The framework should offer flexibility to adapt to changing project requirements and provide extensibility mechanisms to integrate custom modules or components seamlessly.
3. **Community and Ecosystem:** A vibrant developer community and a rich ecosystem of plugins, libraries, and tools can significantly expedite development and troubleshooting processes.
4. **Security:** Robust security features and mechanisms, such as authentication, authorization, and data encryption, are vital for safeguarding user data and system integrity.
5. **Documentation:** Comprehensive and up-to-date documentation simplifies onboarding for developers and ensures efficient utilization of the framework's features.

6. **Learning Curve:** The ease with which developers can grasp the framework's concepts and become proficient in its usage is a critical factor in project timelines.
7. **Cross-Platform Compatibility:** The framework's ability to support various platforms, including web, mobile, and desktop, can influence the project's reach and accessibility.
8. **Community and Industry Support:** Recognition and support from the broader technology community and industry can signify the framework's long-term viability.
9. **Cost Considerations:** Assessing the total cost of ownership, including licensing fees, hosting expenses, and development time, is crucial for budget-conscious projects.
10. **Updates and Maintenance:** Regular updates and a clear maintenance roadmap demonstrate the framework's commitment to staying current and secure.
11. **Performance Optimization:** Built-in tools for performance monitoring and optimization help ensure the application runs efficiently, even under heavy load.
12. **Integration Capabilities:** The ability to seamlessly integrate with other services, databases, and APIs is vital for creating a cohesive and feature-rich application.

These evaluation criteria lay the foundation for a systematic and informed comparison of different application frameworks in the subsequent sections, allowing us to make data-driven decisions regarding the technologies employed in the *SHIDA* project.

III.5 Technology Selection

The *SHIDA* project team prioritizes open-source solutions for transparency and control over the code. Self-hosting reduces dependency on external networks and enhances scalability and security. The team also leverages benchmarks and industry best practices to build upon proven foundations while incorporating cutting-edge elements. Additionally, the team prioritizes stability, maintainability, and wide support in the technologies it selects. This ensures reliable systems and fosters an efficient development process.

III.5.1 Diagram Tools

Diagram tools such as MermaidJS, Draw.io, and Excalidraw were used to create diagrams to help illustrate the various aspects of the project methodology.

For example, a MermaidJS diagram might be used to create programmatically a sequence diagram of user interaction with a component, while a Draw.io diagram might be used to create a diagram of the project's architecture. Excalidraw might be used to create a more informal diagram, such as a sketch of the project team's workflow.

Using diagram tools to illustrate the project methodology has a number of benefits. First, it can help to make the methodology more understandable and accessible to all stakeholders. Second, it can help to identify any potential problems or areas of confusion early on in the project. Third, it can help to communicate the project methodology to new team members and stakeholders.

III.5.2 Setting up Jira and Confluence

Jira and Confluence are two popular tools, described in Section II.16.2, that can be used to support the project methodology described in Section III.2.

Jira and Confluence can be used together to support the project methodology in a number of ways. In this example, Jira is used to define and track the progress of tasks and issues, while Confluence is used to document the project plan and requirements. Confluence is also used to create and share diagrams that illustrate the system.

III.5.3 Software Anatomy Diagrams

Software anatomy diagrams are visual representations of the different components of a software system. They serve to illustrate the relationships between these components, as well as the flow of data and control within the system.

Software anatomy diagrams find applications in various software methodologies, including:

- In the Agile methodology, they are utilized to create user stories and acceptance criteria, ensuring clarity and completeness.
- In the Waterfall methodology, they aid in crafting system design documents and detailed design documents, assuring completeness and accuracy.

- In the DevOps methodology, they are employed in crafting infrastructure diagrams and deployment diagrams, ensuring robust and scalable infrastructure and deployment plans.

Software anatomy diagrams play a crucial role in various aspects of PI planning:

- **Communicating the vision for the PI:** Software anatomy diagrams offer stakeholders a high-level overview of the system that the team plans to deliver during the PI.
- **Identifying and mitigating dependencies:** Software anatomy diagrams help in identifying potential dependencies between system components, enabling the creation of dependency management plans to reduce the risk of delays.
- **Estimating effort required:** These diagrams aid in estimating the effort needed to implement various system components, facilitating the creation of a realistic PI plan.
- **Tracking progress during the PI:** Software anatomy diagrams provide a means to monitor the team's progress throughout the PI, enabling the identification of potential issues and adjustments to the plan as needed.

III.5.4 Code Organization

The *SHIDA* project used a private cloud-hosted GitLab monorepository with strict code rules and code reviews to handle code organization and CI/CD pipelines. This approach has a number of benefits, including:

- **Reduced complexity:** A monorepository simplifies the branching strategy, as all code for the program is stored in a single repository. This makes it easier to keep track of changes and to ensure that all teams are working on the same version of the code.
- **Improved efficiency:** CI/CD pipelines can be used to automate the build, test, and deployment process for the entire program. This can help to improve the efficiency of the development process and to reduce the risk of errors.
- **Increased quality:** The strict code rules and code reviews help to ensure that the code is of high quality. This can help to prevent bugs and improve the overall quality of the program.

- **Reduced duplication:** The use of a monorepository helps to reduce code duplication. This is because all of the code for the project is stored in a single repository. This makes it easier to find and reuse code, and it can also help to reduce the overall size of the codebase.

It is worth noting that configuring CI/CD automation can be expensive for a small team, but the *SHIDA* team invested time in doing so because they believed it would pay off in the long run.

Below follows a more detailed explanation of the three CI/CD pipelines that were defined for the *SHIDA* project.

The dev pipeline is responsible for building and testing the code for the program. It is triggered whenever a change is pushed to the development branch of the monorepository. The dev pipeline runs a suite of unit tests and integration tests to ensure that the code is working as expected.

The test pipeline is responsible for deploying the code to a test environment and running a suite of end-to-end tests. It is triggered whenever the dev pipeline succeeds. The end-to-end tests simulate real-world usage of the program to ensure that it is working as expected.

The production pipeline is responsible for deploying the code to the production environment. It is triggered whenever the test pipeline succeeds. The production pipeline uses a blue/green deployment (explained in Section II.16.4) approach to minimize downtime.

An initial draft of what could be the folder structure of a monorepo is shown in Appendix C.

Monorepos are usually split between applications and packages. Applications are the final products that are deployed to production, while packages are libraries that are used by applications. In the *SHIDA* monorepo, the applications are the frontend and backend, while the packages are the shared libraries. A monolithic repository is also an opportunity to uniformize development tools and processes, such as linting, formatting, and testing.

The organization shown in Appendix C splits the applications into three layers: the presentation layer, the business logic layer, and the data access layer. The presentation layer, where the client applications and UI library is found, is responsible for displaying information to the user and receiving input from the user. The business logic layer, where the backend and its documentation are located, is responsible for processing the user's input and generating output.

The data access layer, where the schemas for data are defined, is responsible for database migration and transactions.

III.5.5 Test framework

The *SHIDA* team chose to use Playwright for their end-to-end tests. Playwright is a relatively new end-to-end testing framework, but it has quickly become popular due to its ease of use and its support for multiple browsers and platforms. For unit testing, Jest was picked in the JavaScript context.

The *SHIDA* project intended to build a dedicated test framework from scratch to improve the quality, efficiency, and maintainability of their testing process. A dedicated test framework could be tailored to the specific needs of the *SHIDA* project, which would help to improve the quality of the tests. It could also help to reduce the duplication of code and improve the efficiency of the testing process by providing features such as automatic test discovery and execution.

In addition to these general benefits, a dedicated test framework could also be used to test the complex interactions between the different components of the system, test the system under different load conditions, and test the system for security vulnerabilities.

Overall, building a dedicated test framework from scratch is a significant investment, but it can have a number of benefits for the *SHIDA* project. By carefully considering the needs of the project and designing a test framework that meets those needs, the *SHIDA* team can improve the quality, efficiency, and maintainability of their testing process.

III.5.6 Human Resources Organization

The *SHIDA* project used a human resources organization methodology that was designed to be flexible and adaptable. The team was divided into two main groups: backend and frontend. The backend team was responsible for developing the server-side logic of the application, while the frontend team was responsible for developing the user interface.

All of the engineers on the *SHIDA* project were expected to work as full-stack engineers, which means that they were able to work on both the backend and frontend of the application. This was encouraged by the team leaders, who wanted to ensure that all of the engineers had a deep understanding of the entire system. Full-stack engineers can reduce duplication of work by working on both the backend and frontend of the system in a vertical cross-section.

The first batch of developers on the *SHIDA* project were picked straight from university, either finishing their degrees or being in a summer internship. This was because the team leaders wanted to hire young engineers who were eager to learn and were determined enough to work on a project that was still in its early stages.

The second batch of developers on the *SHIDA* project were picked remotely from multiple countries in Central Africa after failed attempts to hire from India. They were young but more experienced (associate level) than the first batch. First, the target demographic of the project was in Africa, and the team leaders wanted to hire engineers who were familiar with the African market. Second, the team leaders believed that young engineers from Africa would be more motivated and passionate about working on the project. Third, the team leaders took the opportunity to create a diverse and inclusive team.

The *SHIDA* project's human resources organization methodology was successful in a number of ways. First, it allowed the team to work on multiple features simultaneously and to ship new features quickly. Second, it ensured that all of the engineers had a deep understanding of the entire system. Third, it allowed the team to hire young and talented engineers from Africa.

During the hiring process a number of profiles was done to help search for specific skills on the candidates, as shown in Figure 8. One highly valued skill was experience with real-time communications, since none of the team members had experience with it.

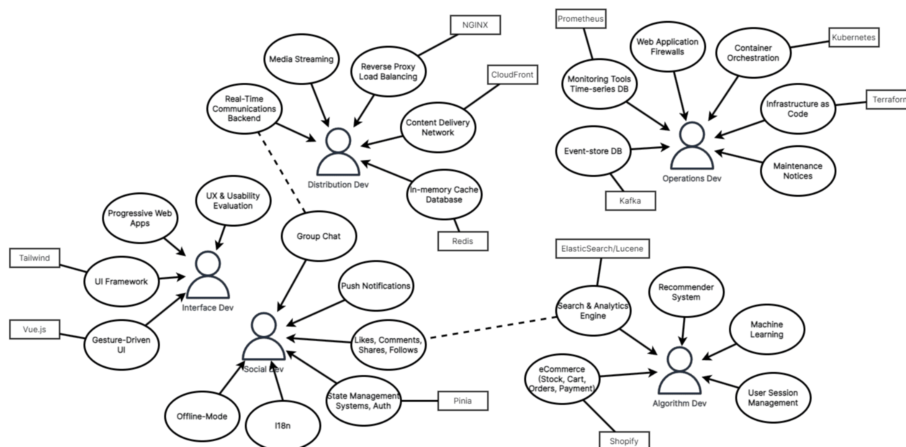


Figure 8: Suggested profiles for hiring

III.5.7 The definition of done

The definition of done (DoD) is a set of criteria that must be met before a task in software development can be considered complete. The DoD is typically defined by the team and is documented in the team's workflow or process.

The DoD is important because it helps to ensure that all tasks are completed to a consistent standard. This makes it easier for the team to track progress and to identify and resolve any problems early on.

For the *SHIDA* project tasks, the DoD consists of a list of acceptance criteria that must be met before the task can be considered complete. This means that the code must be written according to the team's coding guidelines, it must be documented, and it must pass all of the unit tests.

The DoD also states that only tasks with met acceptance criteria are allowed to stage for a commit. This helps to ensure that the codebase is always in a working state and that any commits made are of high quality.

Some more benefits to using a DoD include making it easier to track progress and to identify and resolve any problems early on, reducing the risk of regressions and improving communication and collaboration within the team.

III.5.8 Coding guidelines

Coding guidelines are a set of rules that developers must follow when writing code. They are typically defined by the team and are documented in the team's workflow or process.

The ideal coding guidelines are either enforced by the compiler or are automatically checked by a linter. This helps to ensure that all code is written according to the guidelines and that any violations are caught early on.

The *SHIDA* project team has defined a set of coding guidelines that all developers must follow. These guidelines include semicolons at the end of each statement, indentation with tabs instead of spaces, camelCase for variable names, and others.

III.6 Architectural Choices

The *SHIDA* project team chose to use a cloud-based architecture for their application. This decision was made for a number of reasons, the most critical of

them all being the lack of resources to host the application on-premises. The team also considered the following factors:

- **Scalability:** Cloud-based architectures are highly scalable, allowing the application to scale up or down as needed.
- **Availability:** Cloud-based architectures are highly available, ensuring that the application is always accessible.
- **Security:** Cloud-based architectures are highly secure, protecting the application from malicious attacks.
- **Cost:** Cloud-based architectures are cost-effective, allowing the team to pay only for the resources they use.
- **Maintenance:** Cloud-based architectures are easy to maintain, reducing the team's workload.

Appendix B illustrates what the architecture design of the system looks like, without technology choices attached.

III.6.1 Comparing Clouds

When selecting a cloud service for the *SHIDA* project, the team considered several essential factors. The aim was to find the ideal combination of features, cost-effectiveness, and support.

AWS emerged as the preferred choice due to its optimal blend of features and cost-efficiency. Moreover, it enjoys significant popularity among developers, ensuring ready access to support when required. The extensive range of supplementary tools and services provided by AWS also translates into valuable time and cost savings during development.

It is acknowledged that AWS can be somewhat intricate to navigate due to the sheer variety of services offered, but the team believes the investment in learning will be worthwhile.

In contrast, GCP was not selected due to reported issues with support and high pricing. Additionally, concerns arose from the observation that Google does not use GCP for its own products. To use them is a practice called dogfeeding – a best practice in the software industry.

Microsoft Azure was also considered but ultimately discarded due to concerns regarding potential vendor lock-in and elevated costs.

In conclusion, AWS was deemed the most suitable choice for meeting the *SHIDA* project team's requirements. It offered the optimal balance between features, support, and cost-effectiveness.

III.6.2 Data Access Layer

In crafting the data access layer, the primary focus was on ensuring data ownership while adhering to stringent data protection regulations like GDPR. This led to the selection of open-source software and opting for a self-hosted solution, even if not on-premises.

Embracing the Power of Relational Data

Contrary to the perception of complexity, relational data offers a robust foundation for data management. The utilization of SQL databases not only ensures ACID compliance but also allows for the enforcement of constraints and facilitates the normalization process. Even when dealing with seemingly simple document-based data, an object-relational database like PostgreSQL can be leveraged to enhance data organization and integrity.

It's crucial to recognize that data for a complex system like this extends far beyond mere documents. Object-relational databases prove valuable in handling structured data efficiently. By employing PostgreSQL, the *SHIDA* project team was well-equipped to manage and manipulate diverse data types seamlessly.

Supporting Unstructured Data

For unstructured data, such as caching, analytics and logs, a key-value store emerges as a pragmatic choice. In line with this, the team opted for Redis, which excels in handling such data and provides swift access when needed.

For blob storage, the team considered Amazon S3, but ultimately decided to use MinIO, an open-source alternative. This decision was made to avoid vendor lock-in and to ensure full control over the data.

Neo4j was considered for social networking features, but ultimately discarded due to the team's lack of experience with graph databases and to constrain the scope of the project.

Cloud-native Solutions

The architectural decisions also contemplated the convenience offered by Database as a Service (DBaaS) providers. AWS, for instance, extends support through Amazon RDS for PostgreSQL and Amazon ElastiCache for Redis.

While hosting these services on an EC2 instance is technically possible, it's generally discouraged due to the intricacies involved in managing both the database and the machine.

During the prototyping phase, the team made the most of PlanetScale (based on MySQL) and Upstash (based on Redis/Kafka)'s free tiers, allowing for experimentation and validation of concepts in a controlled environment.

ORM Layer

ORMs, previously detailed in Section II.10, are a popular choice for implementing the Data Access Layer (DAL) in software development, even if not without their drawbacks. The ORM provides a layer of abstraction between the application code and the database, which is expected to make it easier to develop and maintain the application.

There was an intention to accomplish typesafety and well-defined types by mapping database tables and columns to object-oriented classes and properties. There was also the intention to generate migration scripts to easily update the database schemas to new versions.

Database schemas that define the model for each service were created using the Drizzle ORM library, which is a TypeScript alternative to Prisma ORM, which has its own syntax.

Data Model

Appendix A shows a draft of the database schema for the *SHIDA* platform. The schema was designed to be as realistic as possible, hence the dotted lines for the relationships between the entities.

III.6.3 Business Layer

The default approach picked for the business layer was to use a microservices architecture. This approach was chosen because it allows for the different services to scale independently, which is important for a platform like *SHIDA* that is expected to have a large number of users.

The microservices architecture also allows for the services to be slowly implemented one by one, which is important for a project like *SHIDA* that is expected to have a long development cycle.

Furthermore, this architecture allows for the services to be implemented in different languages, which is important for a project like *SHIDA* that is expected to have a diverse team of developers.

The microservices are accompanied by an API gateway that handles authentication and authorization, as well as routing requests to the appropriate service. There was consideration in developing this from scratch, but ultimately the team found and decided to use Kong, an open-source API gateway.

Real-time Chat

The tools found to be most suitable for implementing real-time chat were Socket.IO for text and WebRTC for audio and video.

To solve the need for WebRTC specific architecture, the team considered using Twilio or AWS Chime. Reinventing the wheel does not seem like a plausible solution even considering the possible vendor lock-in and loss of control over the data.

Content Management

Strapi was chosen as the CMS for the *SHIDA* project. Strapi is an open-source headless CMS that provides a flexible and extensible content management solution.

Strapi allows for the schema for “Knowledge Hub” articles, “Book” and “Product” metadata, for example, to be defined. This makes it easy to manage the content for these entities and to ensure that it is consistent across the application. The idea is to allow the content creators to have a simple interface to manage the content.

The CMS being headless allows for the content to be accessed via a REST API, which can be consumed by the microservices layer before being presented to the user. This allows for the content to be reused across multiple services, reducing duplication of effort and ensuring consistency.

E-Commerce

The approach to solve e-commerce was hosting Medusa, an open-source e-commerce solution, as one of the services. Medusa was chosen because it is open-source, which means that it can be customized to fit the needs of the project. Since the architecture allows for each service to be independent in its technology choices as long as it follows the API contract, Medusa fit well. Medusa also has extensions that can be used to add additional functionality to the platform, such as payment processing and shipping. For these, external services like Stripe and Shippo were considered, though no final decision was made.

The adversion to build from scratch was due to the complexity of e-commerce, which is not the focus of the project. The team also considered Shopify, but discarded it due to the lack of control over the code.

Authentication

The team considered using Auth0, an authentication as a service provider, but ultimately decided to implement their own authentication service. This decision was made because the team wanted to have full control over the authentication process and to ensure full responsibility over user data.

III.6.4 Presentation Layer

The *SHIDA* project aims to take the advantages of both client and server-side rendering to the cases where they're most suitable. The team ultimately landed on Next.js as the most feature-packed framework to deal with the frontend.

The most important choice done for the *SHIDA* frontend is focusing on a Progressive Web App (Section II.3.1) approach. This allows for the frontend to be installed as an application on the user's device, which is ultimately takes a website and turns it into a faux cross-platform application. It also allows for the frontend to be used offline, which is important for a platform like *SHIDA* that is expected to have a large number of users in areas with poor internet connectivity. The implementation of a dedicated native app is still considered, but left back for a later stage of the project.

The approach to design was to separate the design tokens and reusable React components into a separate library, which is then supported by Storybook to allow for the components to be tested in isolation. This allows for the components to be reused across the different services, reducing duplication of effort and ensuring consistency. This approach requires a lot of care in making sure that the components are generic enough to not be bound to the pages' implementation, for example by using props to pass data to the components.

Book Reader and Audiobook Player

Readium[32] is one of the most straightforward ways to implement a book reader. It is an open-source library that can be used to render EPUB files in the browser. However, the team considered that it was important to implement a custom book reader to allow for the maximization of value proposition and user experience.

[32] https://readium.org/

As for audiobook player, the choice is between having the audio streamed in chunks from a server or having the audio downloaded in full while protected by DRM. Each solution has its pros and cons. An open source start point considered was Howler.js[33]. The team also considered that it was important to implement a custom audiobook player for the same reasons as the book reader.

[33] https://github.com/goldfire/howler.js

IV Development

This chapter discusses the various technologies and tools that were used to build the *SHIDA* project. It covers the different iterations of the frontend, backend, and infrastructure, as well as the communication and collaboration tools that were used by the team.

IV.1 Preliminary Phase

During the preliminary phase, the vision was shared, and a preliminary plan was made to outline the main priorities and goals for the first weeks.

The project documentation was started by developing an anatomy diagram, its purpose specified in Section III.5.3. The diagram was obtained through a long meeting where all the parts the system is dependent on were first put into a cloud and then slowly wired together. The diagram was then used as a reference point for the rest of the project.

IV.2 API Development

The *SHIDA* project's API has gone through several iterations throughout the development process. Each iteration has brought new perspectives to the API's architecture, performance, and security.

IV.2.1 Iteration 1: Express.js and Docker Compose

In the first iteration of our API development, the team adopted Express.js along with Docker Compose. This choice allowed us to realize that there was often no need to reinvent the wheel when building the API. There were designs at this point to build the whole schema from scratch, which was would've been a one-way ticket to a lot of technical debt.

At this point in time there were also plans to prioritize GraphQL, since it would allow the backend and frontend teams to work more harmoniously, without needing to constantly redefine the endpoints depending on the needs of the frontend.

IV.2.2 Iteration 2: NestJS

The second iteration introduced NestJS, a framework that brought its own challenges. We faced difficulties integrating Medusa, encountered a high degree of opinionation, and encountered significant boilerplate code. This iteration prompted us to question the essence of microservices.

The challenges imposed by Medusa were its dependency on Express.js and TypeORM. A project fork would be necessary if the goal was to umbrella the

whole backend under NestJS. NestJS itself also follows a very opinionated approach, which is not necessarily a bad thing, but it can be a challenge when trying to integrate it with other frameworks and turn into complexity later. The boilerplate code also turned into a challenge, as it made the code readability suffer.

One advantage from NestJS that is missed is the ability to define the communication channels between microservices, using in this case Redis.

IV.2.3 Iteration 3: Next.js API Serverless Functions

For our third iteration, we explored the use of Next.js API serverless functions. While this approach appeared promising, it came with limitations. The challenge of managing and splitting an API gateway arose, making us reconsider its feasibility.

Above all, the main reason for considering this approach was the possibility of distributing the API on the network edge, attached to the frontend, which along with reducing latency and improving performance (except in the case of cold starts) would also reduce complexity associated with managing frontend and backend separately.

The fact that it would allow us to use the same language for both the frontend and the backend was another selling point. This would make it easier to share code between the two, and it would also make it easier to onboard new developers, as they would only need to learn one language.

Overall, the team realized that using these serverless functions seems more valuable for small and simple projects that don't require a lot of customization, especially on the security aspect, as it would be hard to implement a lot of security features that are usually provided by API gateways.

IV.2.4 Final Iteration: Django and Medusa

In our final iteration, we transitioned to using Django along with Medusa[34]. This choice was guided by the preference of some team members for a more familiar and comfortable environment.

[34] https://github.com/medusajs

Until this point, development was done in-house in Gothenburg, Sweden, mostly by a single person. From this point on, the team started growing and the project was moved to a remote-first setup.

The goal for Django was to take advantage of its authentication system to build the high-priority user management features.

Medusa is also deployed as a big service more than a microservice, since it groups all of the commerce-related features into a single service. This is a good approach for a small team, as it reduces the complexity of managing multiple services, but it can become a challenge as the project grows.

Medusa provides more than just the commerce backend. It also provides a Next.js storefront template, which was taken as inspiration for the *SHIDA* frontend. Furthermore, it provides an admin dashboard built with React, which can be hooked up to the Strapi CMS for managing products and their information.

IV.3 Web App

The Web App was branded from the start as a PWA, since it solved a lot of the requirements that were set for the project. It would allow the app to be distributed on the app stores, to work offline, and to take advantage of all that the Web has to offer.

IV.3.1 Iteration 1: React Native for Web

Our initial web app development employed React Native. During this phase, we realized that creating a navigation bar was more challenging than it initially appeared. We also realized that Expo, the framework recommended for developing React Native over "bare", has too much control over the development process, in particular the CI/CD pipeline.

Ejecting from Expo is risky and requires a lot of work that could've been done from the start. It doesn't support all libraries, but it threatens the developer with bad DX (*Developer Experience*) if they don't use it, which makes it feel like a trap.

Expo is therefore a good choice for prototyping small apps that don't require a lot of customization, but it's not a good choice for a large-scale project.

It is harder to find updated documentation on React Native for Web, the way the project is structured is not very intuitive, and the interfaces don't feel

customizable enough. Building both iOS and Android artifacts also results in having to test both, on top of the web version, and they can be inconsistent.

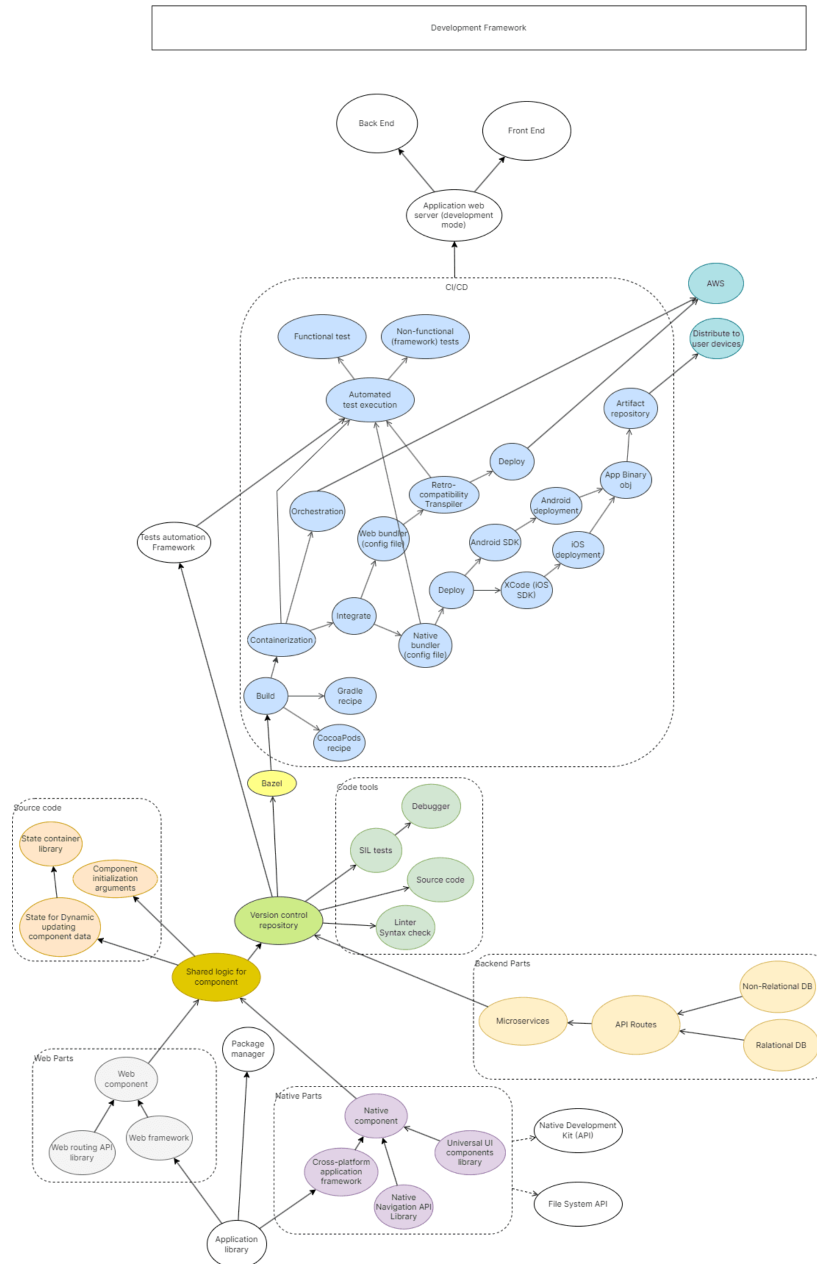


Figure 9: First iteration of the Anatomy Diagram

IV.3.2 Iteration 2: Nuxt

In our second iteration, we explored Nuxt for web app development. Nuxt is a Vue.js framework that provides additional features for building complex web applications. We chose Nuxt because we preferred Vue.js over React. Vue.js had recently been gaining popularity, and many developers were looking for an alternative to React.

One of the main factors in our decision to choose Vue.js was its intuitive new syntax. This syntax made it easier to onboard new developers, but it also made it more difficult to find documentation and libraries that supported it.

Another important exploration of this iteration was the Vite build tool. Vite made it easier to configure and control the build process, and it also made it faster to build and reload the app in development.

The main inspiration for this iteration was the starter template provided by byoungd on GitHub[35], which has the potential to make development a lot faster.

However, we encountered some limitations within the Vue ecosystem. We struggled with customization and configuration issues related to the small selection of modules[36] that were compatible with the new Nuxt 3. Using the older Nuxt 2 version was not an option because it didn't have an edge over other options, namely Universal Rendering, Vite support, etc.

[35] https://github.com/byoungd/modern-vue-template
[36] https://nuxt.com/modules

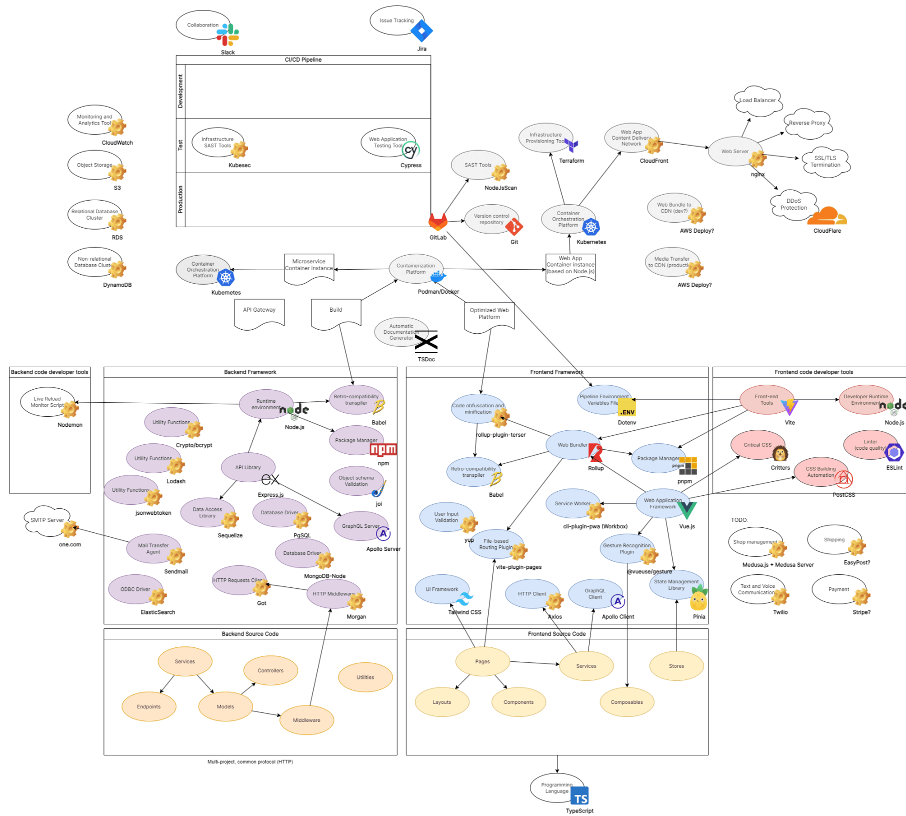


Figure 10: Second iteration of the Anatomy Diagram

IV.3.3 Setting up PWA

During this phase, we ventured into setting up a PWA to enhance the user experience. There are two main options for doing this: use an existing library with a single point of configuration, but be prepared for the possibility that it may not meet all of your needs. Alternatively, configure everything yourself by adding a client-side JavaScript file that runs a service worker and a manifest file that tells the browser how to treat your app. It also needs extra attention to be added as a script to the root HTML template. You then have to manage caching and offline support manually, which needs a lot of planning and testing.

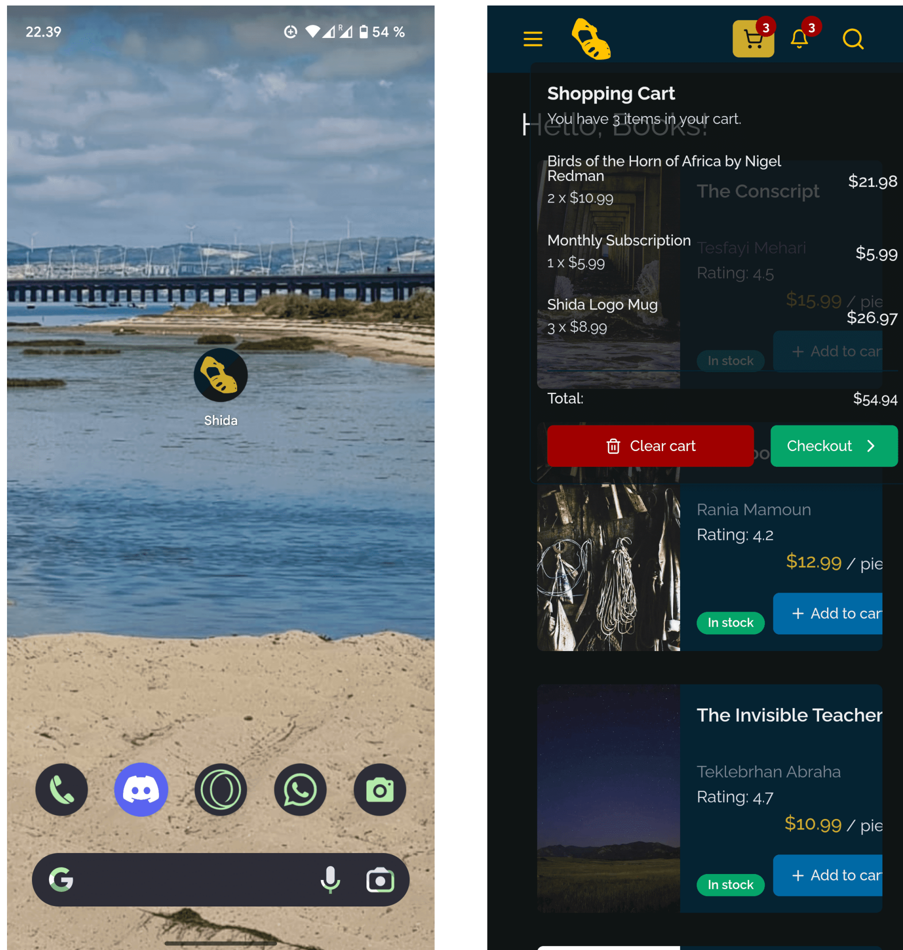


Figure 11: The *SHIDA* prototype, installed as a Progressive Web App on Android (accessible at https://standalone-eta.vercel.app/)

IV.3.4 Setting up Internationalization

The team explored the use of internationalization libraries to make the app available in multiple languages. The most popular options were i18next and vue-i18n, but the team also explored the possibility of using the native browser API, which is not as intuitive.

The way it was done included a locales folder with YAML files for each language, and a plugin that would load the language files and make them available to the app. In the components, the text would be replaced by the term's ID

(for example, `button.login`) wrapped in a `<t>` tag, which would be replaced by the text in the correct language. Then, a language switcher component would be used by the user to change the language.

IV.3.5 Microfrontends Exploration

Our exploration also extended to microfrontends, an approach aimed at breaking down monolithic frontends into smaller, more manageable parts. This approach is particularly useful for large-scale projects with multiple teams working on different parts of the frontend. One of the main challenges is configuring the routing and state management (like keeping the authenticated user) between the different microfrontends, which has a very limited number of existing solutions such as, in this case, Webpack Module Federation. Ultimately, it doesn't feel like a production-ready solution, and the separation between the features feels too drastic.

IV.3.6 Iteration 3: Flutter

Because of its high popularity, the team decided to explore Flutter for one of the prototypes. Flutter is a cross-platform development framework that allows developers to build native-looking apps for iOS, Android, and the web. It was expected to make it easier to implement mobile user experience features, such as swiping and scrolling, and to make it easier to build a consistent design system across platforms.

However, Flutter for web proved to be poorly optimized and introduced a lot of complexity into the project. It also didn't feel like a good fit for the project, as it would require a lot of work to make it work with the existing backend.

While Flutter may have its place, it's important to consider its limitations and suitability for specific application types, especially for new developers who might be drawn to it without a full understanding of its strengths and weaknesses.

IV.3.7 Final Iteration: Next.js

Ultimately, we settled on Next.js as our final iteration for web app development. Next.js is a mature React framework with a large community and a lot of documentation. It also has a lot of built-in features that make it easy to get started, such as routing, server-side rendering, and static site generation.

We used Storybook to develop and document our components. Storybook is a tool that allows you to develop and showcase your UI components in isolation.

We used Auth.js for authentication. Auth.js is a mature library that supports a lot of authentication providers and is easy to configure. It also supports JSON Web Tokens (JWTs), which are a must-have for microservices. However, Auth.js does not extend to the backend, so we needed to use a solution like the one outlined in Section IV.2.3. The later solution using Django made it harder to configure, but it was still possible to use Auth.js for the frontend and Django for the backend.

The team hosted their prototype under Vercel with a free plan. Vercel is a cloud platform that makes it easy to deploy and host Next.js applications. The main reason to not use it in production is that an investment was made in AWS, and it doesn't have as many production-ready features as AWS.

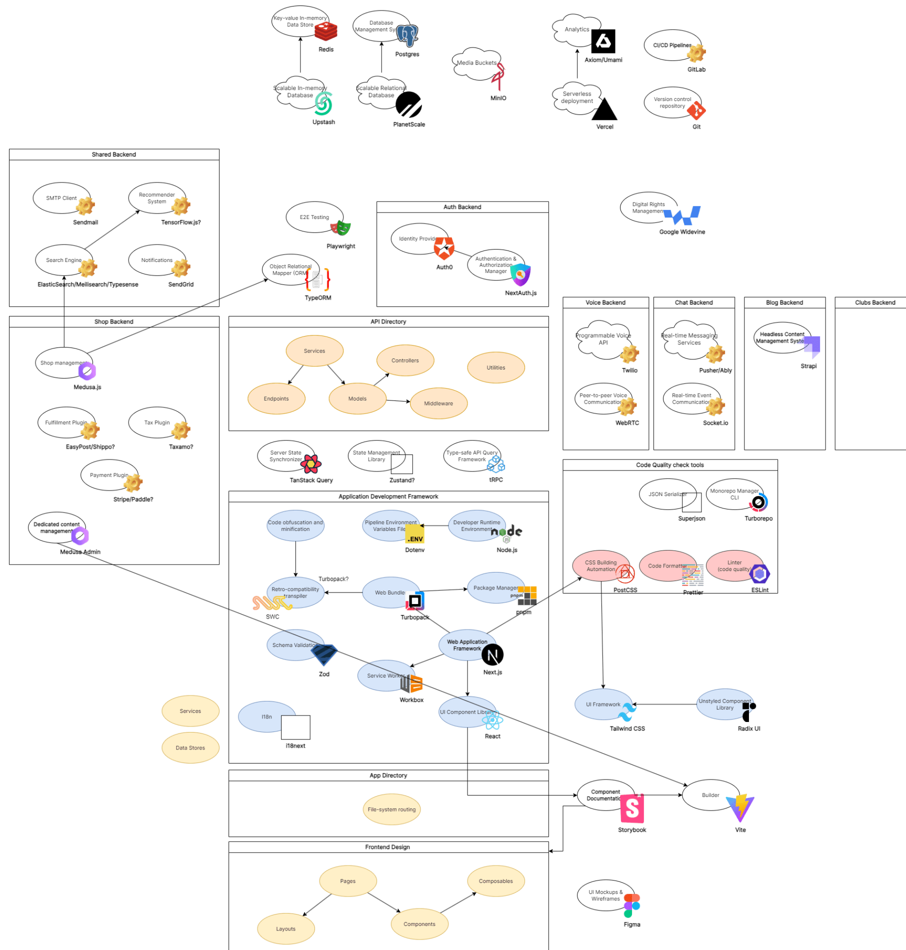


Figure 12: Third iteration of the Anatomy Diagram

Setting up the Design System

We chose Tailwind CSS as our CSS framework because it makes it easy to create a consistent design system and to customize it. We considered using the Flowbite component library to complement Tailwind CSS, but we decided that it was not worth the investment due to its pricing model. Instead, we chose to use Preline in production.

Preline UI is a new component library that is getting a lot of traction in the community because of its copy and paste approach. Other component libraries that we considered were NextUI, which is a bit more opinionated but still built

on top of Tailwind CSS, and @shadcn/ui, a new component library that uses a copy-and-paste approach.

Figma was used to make some prototypes of the UI and also some branding exploration. Lack of experience with the tool and unwillingness to invest early in a paid plan led to a lack of depth in these, but they were still useful to get a sense of the design direction. Some help was provided by collaborating with a team from Uptive[37], a company that specializes in UI/UX design. These prototypes implemented in Figma are shown in Figure 13.

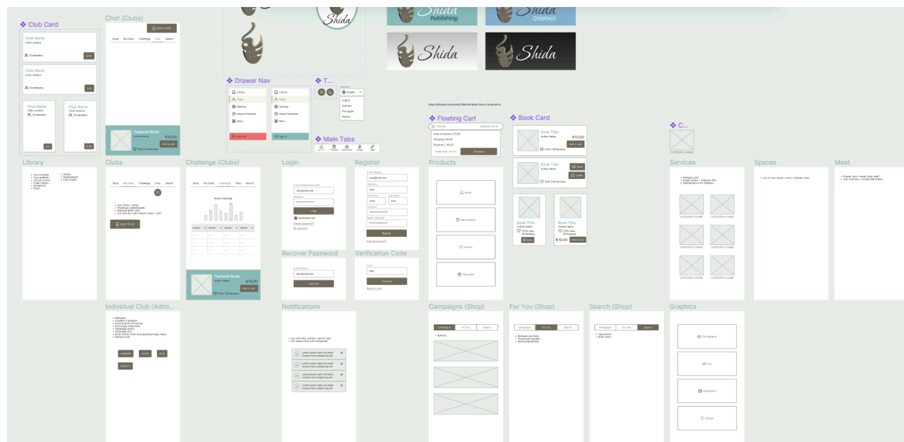


Figure 13: The *SHIDA* prototypes, implemented in Figma

There was already a decision made to not stick to the current state of design. As such, a redesign from scratch by experienced UI/UX designers with official branding is on course.

Implementing the Interface

To determine routing for the web app, the team used a flowchart to map out the user's navigation through the website. This flowchart, shown in Figure 1.6, was used as a reference point for the rest of the project.

[37] https://www.uptive.se/

Figure 14: Diagram of the user's planned navigation through the website

On the latest prototype, as seen on Figure 11, a few components were built up for display. The main app shell consists of a header that can toggle a sidebar. This header has a variably colored logo and name on the left, and on the right it features a button that opens a search modal, a cart popover, a notifications popover and some social media links.

The popovers had some work put into them so they could show a small badge with the number of items in the cart or unread notifications, and so their hover state would be consistent with the rest of the app.

The cart popover also features a list of items with their name, price, quantity and a button to clear the cart. The cart clearing button features a confirmation modal, which is good UX practice for destructive actions.

The notifications popover features a list of notifications with their title, description and a button to mark them as read. Even after clearing, these notifications are supposed to be all accessible in a separate page. This popover also has a switch built into it that would allow the user to opt into or out of push notifications.

The search modal features a search bar with a button to toggle between searching for products and searching for users. It also features a list of results with their name, price and a button to add them to the cart. The search modal is designed to work throughout the whole website: the results are separated into categories such as "Books", "Clubs", "People", "Articles", "Talk Rooms", etc. The talk room results have a small badge that indicates their status as "Live", date and time they're scheduled to, or "Done".

The sidebar features a list of links to the main pages of the website, and a display for the user's profile, settings, and login or logout. This login button includes the state management for the user's authentication, though it just

toggles between a guest and a dummy user for now. One feature explored with these buttons was the microinteractions that should happen on any potentially expensive action, such as switching the button's state to disabled with a spinner while log in is being processed or the logout confirmation modal is open.

The settings bar also features a modal where the user can switch between dark and light themes, a dropdown for interface language, planned to be integrated with a library, and a dropdown for the store currency, which is information obtained through the e-commerce API.

In the books page, the user can see a grid of book cards, each with their cover, title, author, price, and a button to add them to the cart. Figure 15 shows these book cards and how they can break under certain screen sizes. The layout that was aimed for is less trivial than it seems: the cards are supposed to be responsive, and the grid is supposed to be responsive as well. Furthermore, the images are loaded after the page is loaded, and the images are supposed to be cropped into a specific aspect ratio.

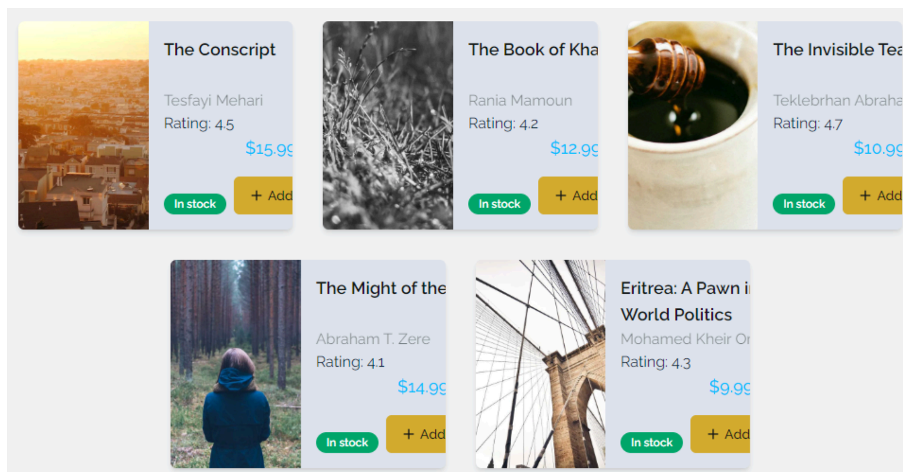


Figure 15: Book card grid: more complicated than it seems

The frontend team was encouraged to model the complex interactions the user can have with each component using diagrams like sequence diagrams. Figure 16 shows an example of this, by using a flowchart to explain the form's state throughout a user's login process.

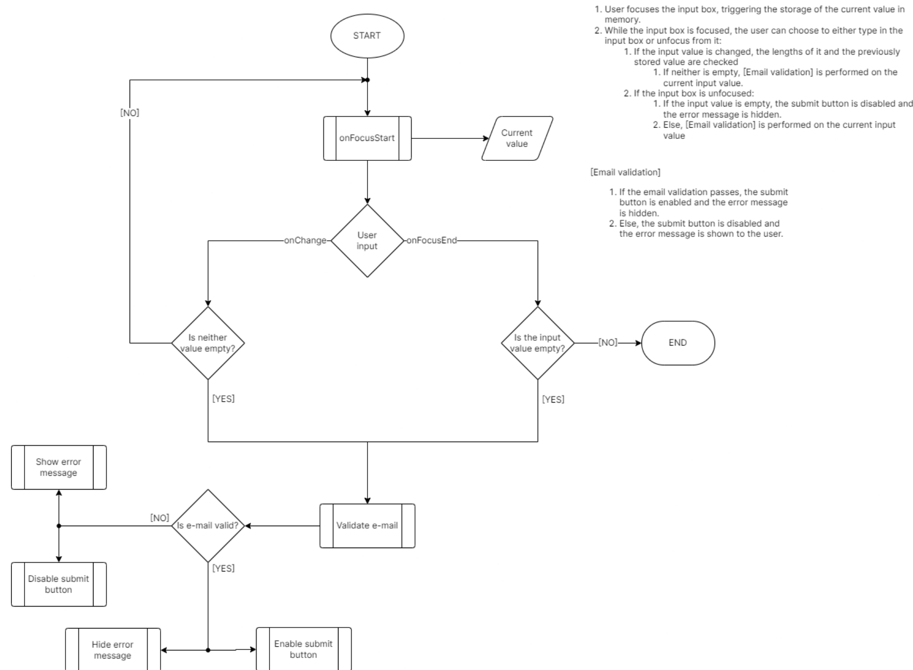


Figure 16: Flowchart explaining client-side dynamic input validation

IV.4 Setting up AWS EC2 and GitLab

AWS EC2 was picked as the option for a virtual machine because of its popularity and the fact that it's the most mature option.

Amazon Web Services has a serious problem with user freedom, mainly when editing the configuration of the services. It's very easy to get lost in the configuration options, and it's very easy to make a mistake that can force a start from scratch. It's also very easy to get lost in the documentation, as it's not very well organized and it's not very clear what the best practices are.

GitLab was picked over GitHub because it can be self-hosted, and thus the source code is protected as mentioned in the requirements. GitLab also has more enticing DevSecOps and CI/CD integration features than GitHub. Worth noting that the version of GitLab installed was actually the enterprise edition. Picking it over the community edition came down to the possibility of easily upgrading to a paid tier in the future, which would allow for more features.

Nonetheless, a virtual machine in AWS EC2 was provisioned to serve as main development environment. This machine had to be expanded later, because the free tier's memory and storage are not enough to host GitLab. The distribution picked was Debian 12 "bookworm", which is known to be stable and easy to use. The reason for picking Debian over Ubuntu is its lighter footprint.

The machine was configured to be accessible through SSH (*Secure Shell*), which in AWS's case requires a key pair, which was generated in the platform. Also, to save up to 76% of the cost, the machine was configured to be turned on only during working hours, and to be turned off during the rest of the day. This was accomplished using two AWS Lambda functions. These are "cron jobs" written in Python that used the AWS SDK to turn the machine on and off at specific hours.

To install GitLab on the virtual machine, the team followed the official documentation[38].

IV.5 Setting up VPC

A VPC (Virtual Private Cloud) can be viewed as a logically isolated network infrastructure that provides control over network configuration and security. It allows developers to create their own private virtual network within the cloud, resembling the traditional on-premises network setup.

A plan was made to define the network and its IP address range, subnets, route tables, and network gateways. The four main subnets were defined as "developer" for builds and developer tools, "data-center" for databases, DMZ for public-facing services, and "services" for the private services that are not exposed to the Internet. The team also defined a Network ACL (*Access Control List*) to control inbound and outbound traffic to and from the subnets.

Figure 17 shows an informal network diagram of the *SHIDA* project.

[38] https://about.gitlab.com/install

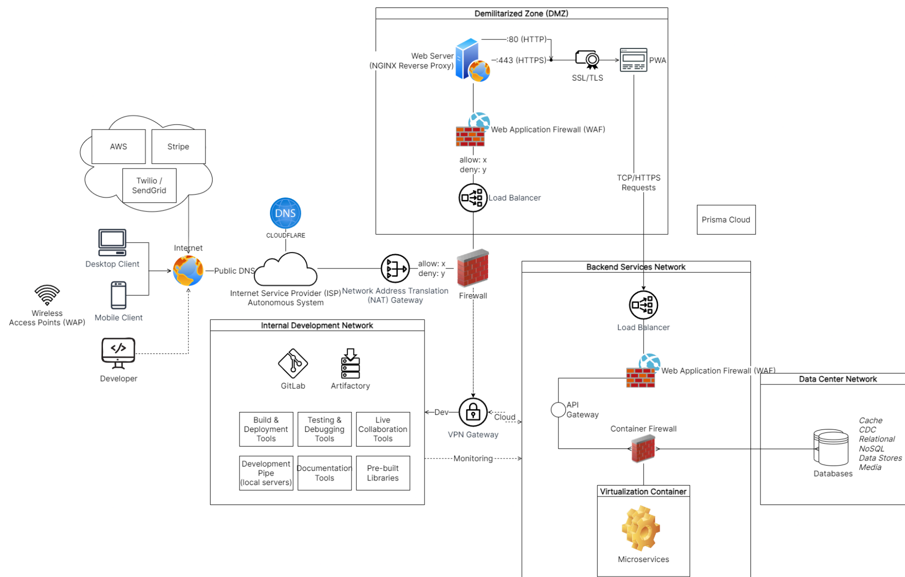


Figure 17: Informal network diagram of the *SHIDA* project

Misconfigurations in this context can lead to connectivity issues, security vulnerabilities, or suboptimal performance. Therefore, comprehensive knowledge of networking features, best practices, and troubleshooting techniques is essential for engineers working with cloud infrastructure.

IV.6 Setting up VPN and Certificates

The team explored the setup of a VPN (*Virtual Private Network*) and Private Certificate Authority (PCA) to enable developers to securely access the GitLab instance.

When not sure about how to use the best security practices, the team arranged a meeting with experts from CGit AB[39], which pointed towards a Palo Alto Networks product called Prisma Cloud[40]. The team talked with sales representatives and ended up quitting the idea because of the high cost and the fact that it would be overkill for the project's current state. The practices that were most focused on were SAST (Static Application Security Testing), DAST (Dynamic Application Security Testing), and protecting IaC scripts

[39]https://www.aixia.se/en/
[40]https://www.paloaltonetworks.com/prisma/cloud

(Infrastructure as Code) using a FOSS (*Free and Open Source Software*) tool called checkov[41].

On the GitLab EC2 instance, the team set up a DNS proxy server using the dnsmasq tool to resolve internal addresses as names. The AWS VPN is built on top of OpenVPN, an open-source VPN solution that uses SSL/TLS for key exchange. The team configured the VPN to use the same certificate authority as the GitLab instance, which was set up using the `easy-rsa` tool, a set of scripts that simplify the creation and management of a private CA. The team used this `easy-rsa` tool to generate the necessary certificates and keys for the developers to use the VPN. The team then tasked the developers with installing an OpenVPN client on their machines and sent them configurations that use the certificates and keys generated by the CA.

In addition to the DNS proxy, the team also requested a public certificate through the certbot tool over the `shidanetwork.com` domain (which required a DNS challenge). This allowed developers to access the GitLab instance by connecting to the VPN and using the `gitlab.shidanetwork.com` address, which has access to the Internet from inside but is not accessible from the outside.

IV.7 Setting up the Monorepo

The team's first attempt to bring all of the project's packages into a single repository used Bazel, the most complete monorepo tool available. However, the team struggled with setting up and configuring this complex tool, especially when using it for web bundling JavaScript. Google and the Bazel team discontinued support for the official Bazel JavaScript rulesets, so a new company called Aspect became the maintainers of the new rulesets. Adapting to TypeScript and other tools was also challenging, and there isn't support for a lot of tools. Finally, requiring every developer to install the Bazel CLI was difficult, as the installation process was not straightforward and the documentation was unclear.

The team then explored Turborepo, a new monorepo tool built by Vercel, the people behind Next.js, but not before trying out Nx and realizing that solution came with its own set of challenges. Turborepo has the great advantage of being part of the same toolchain as new Next.js build tool, Turbopack. Turborepo is very simple compared to the others, and easy to be incrementally

[41] https://www.checkov.io/

adopted: it is simply a layer above any package manager's workspaces, made to run commands like "lint", "build" and "test" a lot faster.

IV.8 Developer Communication

During the duration of this development, and while there was a team, the team used Slack for internal communication. Every Friday, the team had a meeting to discuss the progress of the current Epic and to plan the next sprint. Everyday, however, the team had a standup meeting early in the morning to update each other on the progress of the tasks and to discuss any blockers.

A Jira board of tasks was defined over a few meetings in order to organize priorities and detail the requirements further to make them more actionable. The definition of tasks for the *SHIDA* project started by defining the major goals through Epics, then adding User Stories to them and finally tasks. The board was then used in each standup meeting to track the progress of the project.

Appendix D and Figure 18 show two Excalidraw whiteboards that were used to plan tasks and keep track of the system design.

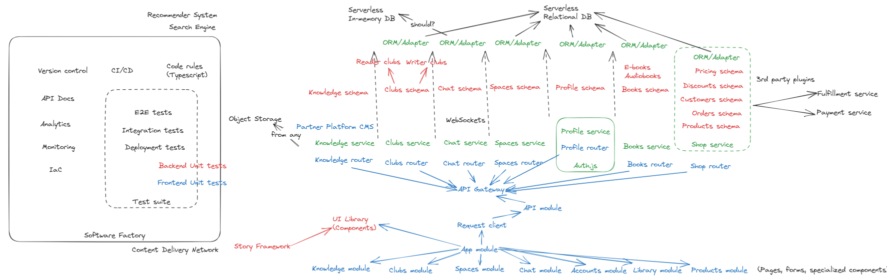


Figure 18: Excalidraw whiteboard used to keep track of system design

V Conclusions and Future Work

This chapter concludes the thesis by summarizing the key findings of the *SHIDA* project, highlighting the challenges and opportunities that emerged throughout the development process. It also explores promising avenues for future research and development in the field of modern software engineering.

V.1 Summary of Findings

Our research journey into the world of modern software development, with the *SHIDA* project as a case study, has revealed a myriad of insights and challenges. The main findings and contributions of this research can be summarized as follows:

Managing Complexity

One of the fundamental challenges encountered in developing *SHIDA* was effectively managing complexity. The project's expansive scope, spanning social networking, e-commerce, e-learning, literature, and more, presented a formidable challenge. It underscored the importance of maintaining a clear vision while remaining adaptable and embracing an iterative approach.

Scope Creep and Unrealistic Expectations

Scope creep emerged as a recurrent theme in the *SHIDA* project, often stemming from unrealistic expectations and inadequate task estimation. The ongoing challenge of scope management necessitates clear requirements and realistic expectations from the project's inception.

The Significance of Research and Analysis

The research and analysis phase, especially when adopting a long-term perspective, demanded considerable time investment but proved indispensable. It underscored the notion that dedicating time upfront to comprehend and meticulously document requirements substantially reduces development efforts. Incomplete or ambiguous requirements frequently resulted in inefficiencies during the development process.

Developer Experience with Technology Choice

As the project evolved, the importance of considering developer experience when making technology decisions became evident. While initial choices were influenced by the state-of-the-art, we learned that prioritizing developer experience can enhance code quality and overall productivity.

Navigating the JavaScript Ecosystem

The abundance of supporting libraries introduced potential incompatibilities and overlaps, hindering development and adding complexity. Employing a more streamlined approach to library selection, such as picking a "batteries-included" tech stack, could have simplified the development process and reduced technical debt.

Embracing a Mobile-Native-First Approach

Exploring a mobile-native-first approach for the frontend could have been a valuable strategy. While maintaining a web presence remains crucial, placing primary emphasis on mobile could have offered users a more tailored experience, particularly given the widespread use of mobile devices for internet access in the target region.

Iterative Development

Navigating the iterative development process introduced a degree of uncertainty regarding the depth of each iteration before reevaluation. Striking the delicate balance between making progress and revisiting decisions emerged as a nuanced challenge.

Network Infrastructure

Our experiences with network configuration underscored the intricacy of establishing a secure and robust production environment. Manual and "quick" configurations, although functional, presented challenges. Future endeavors to automate and optimize network infrastructure for distributed systems hold promise.

Starting at the Highest Complexity

Initiating the project as a relatively inexperienced engineer at the highest complexity levels posed significant challenges. Future research could explore strategies to facilitate developers' entry into complex projects, ensuring a smoother learning curve.

State-of-the-Art and Numerical Analysis

The *SHIDA* project drew heavily from state-of-the-art technologies and practices. Employing or finding meaningful and accurate numerical analysis, guided by objective and realistic metrics, proved a big challenge when comparing these technologies. Future research could explore strategies to evaluate the effectiveness of state-of-the-art technologies in an unbiased manner.

Rapid Technological Advancements

Exploring the integration of cutting-edge frameworks could potentially streamline and enhance project management, particularly within the context of large-scale web applications like *SHIDA*.

Balancing Rigor in UML Diagrams

The exacting rigor demanded in crafting UML diagrams, while advantageous, can be time-intensive. Therefore, it is imperative to consider the trade-off between rigor and efficiency. Future research endeavors could investigate strategies to streamline or automate the process of translating conceptual ideas into UML diagrams, particularly for large-scale projects, or even suggest alternatives to UML.

Performance Issues in Complexity

Contemporary computers boast enhanced speed and processing power, making concerns about performance take a backseat. Nevertheless, increasingly common complex codebases give rise to numerous minor unoptimized performance costs that often accumulate into substantial challenges. The addition of extra processing power doesn't consistently resolve these issues, which can result in users feeling the app is sluggish which leads them to disengage from it.

V.2 Future Directions

While this research has provided valuable insights into various aspects of software development within the context of the *SHIDA* project, it also opens the door to several promising avenues for future exploration:

Further Investigation into Orchestration

Despite its potential benefits, the *SHIDA* project did not reach a stage where Kubernetes could be comprehensively assessed. Subsequent research endeavors should aim to delve deeper into the practical implications and advantages of Kubernetes in managing the complexities of large-scale projects.

Machine Learning Integration

A compelling prospect lies in the integration of machine learning algorithms to enhance user recommendations, personalize content delivery, and facilitate predictive analytics. Such integration could yield improvements in user engagement and content discovery, warranting thorough investigation.

Graph Databases

Exploring the implementation of graph databases is another promising direction. This exploration seeks to optimize complex query execution and minimize database calls, potentially resulting in more efficient data retrieval processes.

Monitoring and Logging

The implementation of robust monitoring and logging mechanisms is essential to ensure the smooth functioning of the platform. Future research could explore the integration of advanced monitoring and logging tools to enhance the project's stability and reliability.

Feedback Mechanisms

Robust feedback mechanisms, user surveys, and advanced analytics should be implemented to continuously gather user insights. Leveraging this data will enable ongoing refinement of the platform, aligning it more closely with user preferences and behaviors.

Continuous Improvement in Localization

As diverse user bases become increasingly prevalent, sustained research into effective content localization strategies is imperative to ensure the creation of inclusive digital experiences.

Evolving Regulatory Landscape

The digital realm is subject to constant regulatory evolution. Keeping abreast of these changes and adapting compliance strategies accordingly will be an ongoing imperative to ensure the project's legal and ethical alignment.

Further Exploration of Developer Experience

In-depth investigation into the influence of developer experience on project success and productivity is warranted. This research may encompass strategies to enhance developer experience and facilitate a smoother learning curve for newcomers to the project.

Exploration of Cutting-Edge Technologies

As technology continues to advance, the exploration of the latest frameworks and tools holds the potential for more efficient and streamlined project management.

Frontend Frameworks vs. Browser Features and APIs

An intriguing avenue involves examining the possibility of moving away from frontend frameworks and harnessing the capabilities of modern browsers through their features and APIs.

Integration with Local Partnerships

Consideration should be given to the integration of local partnerships through the development of administrative dashboards and CMS.

Modularization of the Project

Exploring the division of the project into smaller, autonomous components under a single organizational umbrella offers the prospect of enhanced manageability and scalability.

These future directions, informed by the findings and challenges uncovered in the *SHIDA* project, represent valuable opportunities for further research and development in the dynamic landscape of modern software engineering.

V.3 Conclusion

In conclusion, our journey through the development of the *SHIDA* superapp has been filled with profound lessons, valuable insights, and a deep understanding of the intricate world of modern software engineering. The *SHIDA* project, with its multifaceted nature, has not only revealed the challenges but also the vast opportunities that come with creating comprehensive, large-scale web applications.

Throughout our research, we addressed the core questions and value propositions posed in the introduction, as outlined in Section I.1.4:

Software Architecture (RQ1)

We thoroughly examined the software architecture decisions in the *SHIDA* project, including the choice between monolithic repositories and microservices. Our findings shed light on how these choices influenced the project's ability to meet its specific requirements, offering valuable guidance for similar endeavors.

Agile Methodologies (RQ2)

Our investigation into agile methodologies, such as sprint planning and continuous integration, within the *SHIDA* project highlighted their role in enhancing the project's adaptability and responsiveness to changing user needs. These insights are pertinent not only for practitioners but also for the broader research community, to assess their effectiveness.

Technological Challenges (RQ3)

We delved into the strategies used in the *SHIDA* project to overcome technological hurdles, such as infrastructure limitations and cross-platform compatibility. These insights offer valuable guidance for projects facing similar obstacles.

User-Centric Design (RQ4)

Our analysis of user-centric design principles in the *SHIDA* project underscored its commitment to creating an intuitive and inclusive platform, especially for users accessing it through mobile devices. This approach not only improved usability but also prevented costly future redesigns.

Content Localization (RQ5)

Our analysis of content localization strategies in the *SHIDA* project highlighted the importance of tailoring content and user interfaces to diverse languages and cultures. This strategy has set the platform apart from competitors.

Regulatory Compliance (RQ6)

We explored how the *SHIDA* project handles complex regulatory landscapes and data privacy requirements in the African countries where it operates. This understanding is crucial for projects operating in similar contexts.

While we have made significant progress, it is important to acknowledge that the *SHIDA* project has not yet reached a production-ready state, despite the substantial work put in. This serves as a stark reminder of the immense challenges involved in such ambitious endeavors.

The *SHIDA* project stands as a testament to the ever-evolving landscape of software engineering and web development. It has expanded our understanding of the complexities inherent in such projects and highlighted areas where further research and innovation are needed.

As we move forward, we embrace the lessons learned and the insights gained from the *SHIDA* project. We look to the future with a commitment to advancing the field of software engineering, contributing to the development of innovative solutions, and addressing the challenges that lie ahead.

In closing, we recognize the significance of this research in enhancing our collective knowledge of building a “super-app” from scratch. The *SHIDA* project, with its grand ambitions, has provided a unique opportunity to explore the intricacies of modern software engineering.

Thank you for joining us on this journey through the intricacies of the *SHIDA* project and the vast landscape of modern software development. We hope that you have gained valuable insights, and we encourage you to continue exploring the exciting world of software engineering.

References

[1] T. Garsiel and P. Irish, “How browsers work - Behind the scenes of modern web browsers”. Accessed: Sep. 03, 2023. [Online]. Available: https://web.dev/howbrowserswork/

[2] S. J. Dixon, “Average Daily Time Spent on Social Media Worldwide 2012-2023”. Accessed: Sep. 09, 2023. [Online]. Available: https://www.statista.com/statistics/433871/daily-social-media-usage-worldwide/

[3] Stack Overflow, “Stack Overflow Developer Survey 2023”. Accessed: Sep. 20, 2023. [Online]. Available: https://survey.stackoverflow.co/2023/

[4] Microsoft Learn, “Three-Layered Services Application”, *Enterprise Solution Patterns Using Microsoft .NET*, 2014, Accessed: Sep. 21, 2023. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff648105(v=pandp.10)

[5] Amazon Web Services, “What Is A LAMP Stack?”. Accessed: Sep. 11, 2023. [Online]. Available: https://aws.amazon.com/what-is/lamp-stack

[6] C. Gross, “A Real World React to htmx Port”. Accessed: Sep. 21, 2023. [Online]. Available: https://htmx.org/essays/a-real-world-react-to-htmx-port/

[7] A. Hodakovskis, “Progressive Web Apps you use every day”. Accessed: Sep. 11, 2023. [Online]. Available: https://medium.com/progressivewebapps/progressive-web-apps-you-may-use-every-day-164bfa92c498

[8] S. Lam, J. Li, P.-H. Shih, D. Christensen, and B. Olajide, “System Design 101”. Accessed: Oct. 26, 2023. [Online]. Available: https://github.com/ByteByteGoHq/system-design-101

[9] Next.js, “Loading UI and Streaming”. Accessed: Oct. 24, 2023. [Online]. Available: https://nextjs.org/docs/app/building-your-application/routing/loading-ui-and-streaming

[10] J. Sumner, A. Partovi, and C. McDonnell, “Bun 1.0”. Accessed: Sep. 11, 2023. [Online]. Available: https://bun.sh/blog/bun-v1.0

[11] G. Maggini, “Deciding Between Native and Cross-Platform Mobile Frontend Programming Frameworks”. Accessed: Sep. 10, 2023. [Online]. Available: https://developer.ibm.com/articles/deciding-between-native-and-cross-platform-mobile-frontend-programming-frameworks/

[12] Nrwl, “Monorepo Tools”. Accessed: Sep. 08, 2023. [Online]. Available: https://monorepo.tools/

[13] A. Shellhammer and J. Neel, “The Need for Mobile Speed”. Accessed: Sep. 13, 2023. [Online]. Available: https://blog.google/products/admanager/the-need-for-mobile-speed/

[14] J. Piikkila, “What Is SAFe (Scaled Agile Framework)?”. Accessed: Sep. 23, 2023. [Online]. Available: https://www.atlassian.com/agile/agile-at-scale/what-is-safe

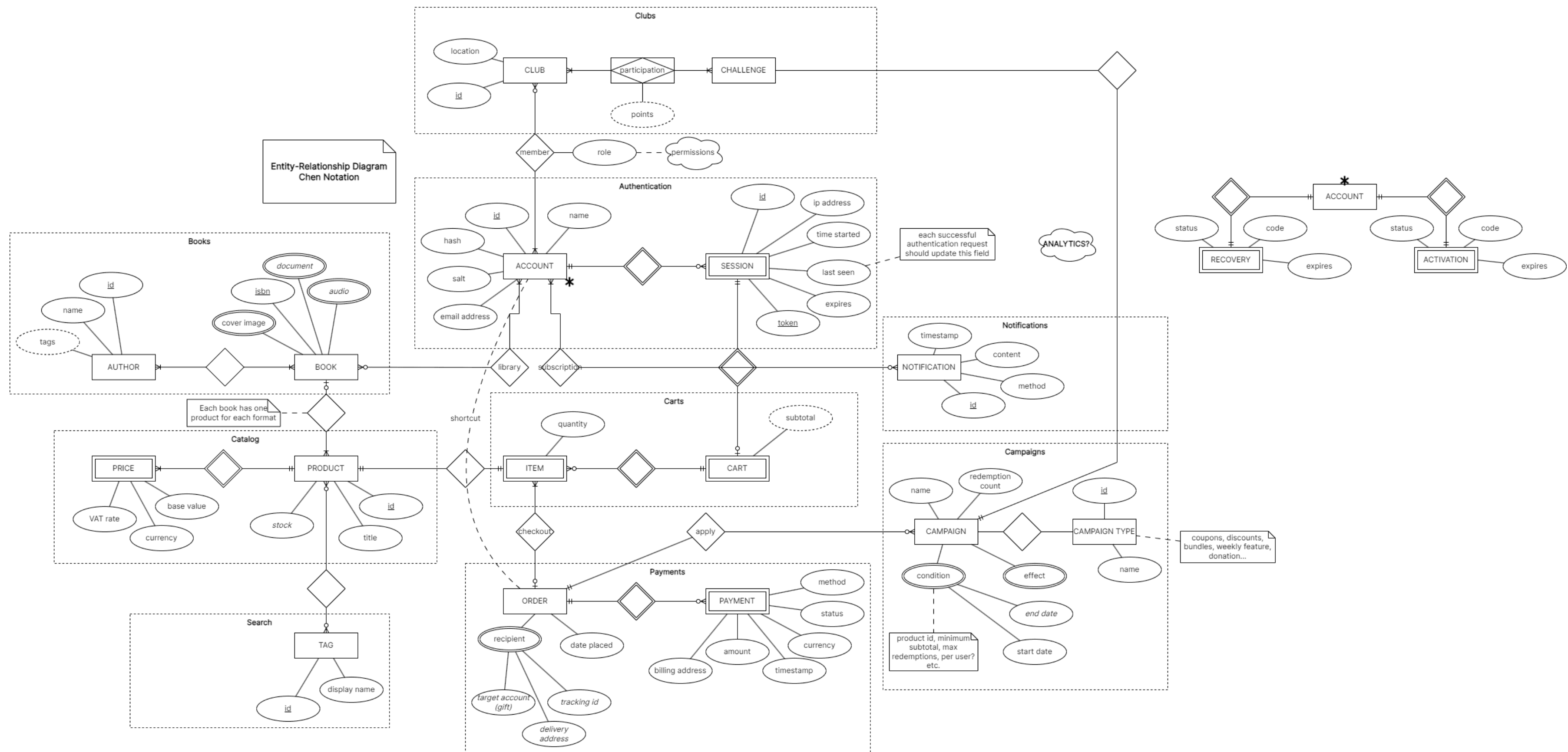
[15] Red Hat, “What is DevSecOps?”. Accessed: Sep. 26, 2023. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-devsecops

[16] M. McRill, “Don't Procrastinate, Iterate”. Accessed: Sep. 04, 2023. [Online]. Available: https://candid.dev/blog/dont-procrastinate-iterate/

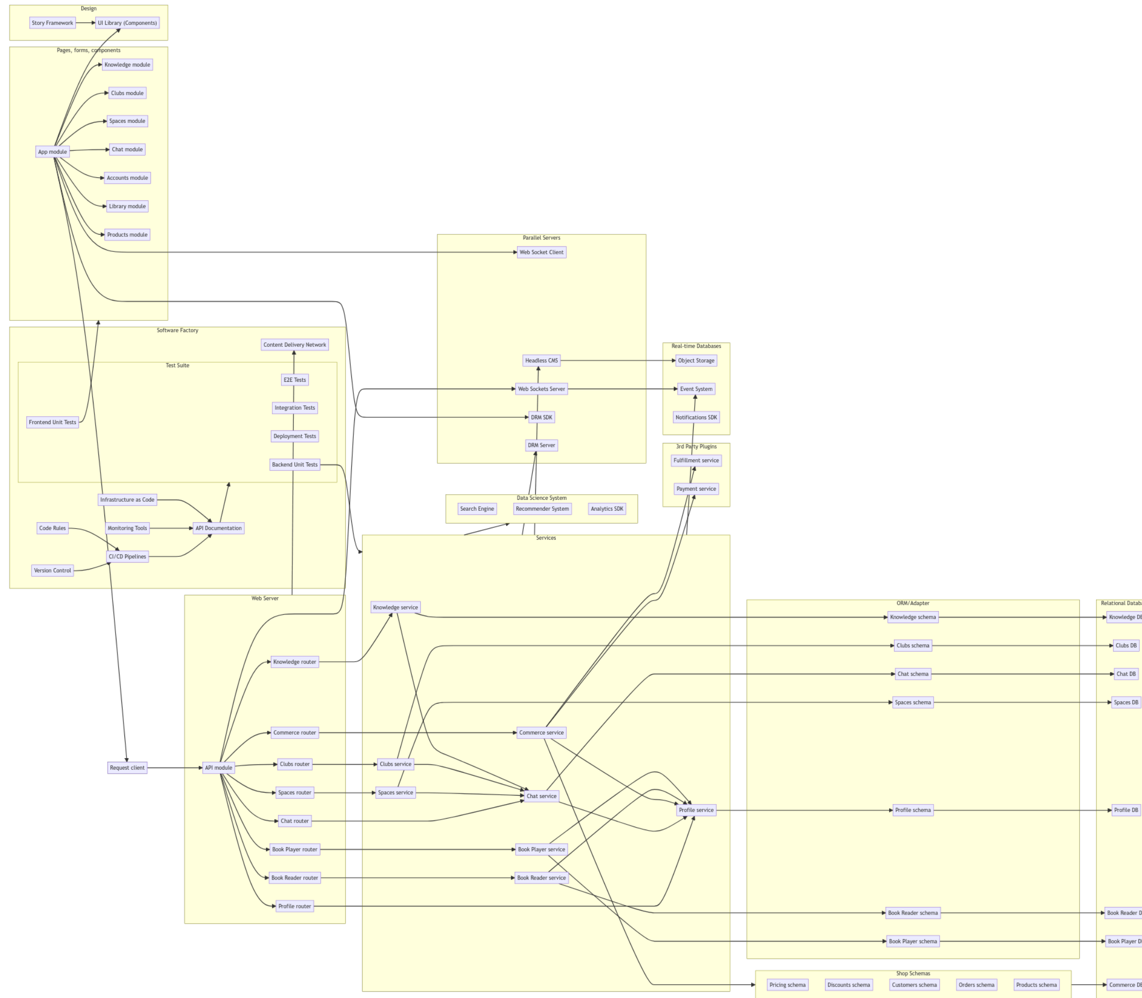
[17] S. Parrish, “Chesterton's Fence: A Lesson in Second Order Thinking”. Accessed: Sep. 12, 2023. [Online]. Available: https://fs.blog/chestertons-fence/

[18] C. Gross, “Complexity Budget”. Accessed: Sep. 17, 2023. [Online]. Available: https://htmx.org/essays/complexity-budget/

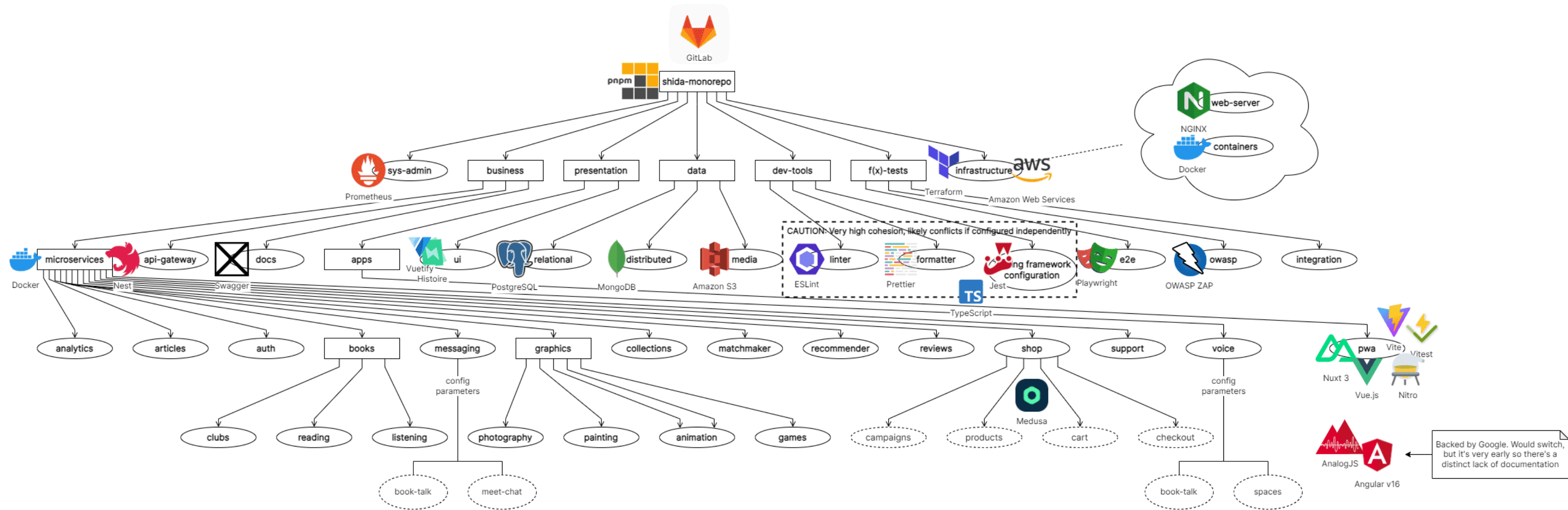
A Entity-relationship diagram



B Functional and non-functional architecture diagram



C Meta-structure for the SHIDA monorepo



D Excalidraw whiteboard used to plan Jira tasks