

POLITECNICO DI TORINO

Master degree course in Digital Skills for Sustainable Societal Transitions

Master Degree Thesis

AI-Assisted Frontend Development of a Monitoring Dashboard for Water Distribution Networks

Relatori
Prof. Gianvito Urgese
Dr. Giuseppe Fanuli
Dr. Walter Gallego Gomez

Candidato
Melikasadat Alamdar

September 2026

Abstract

Water is one of the most essential resources for human life, yet a striking share of it never reaches the people who need it. Global freshwater demand is predicted to exceed supply by 40% by 2030, while an average of 23% of treated water in the European Union is lost during distribution before it even reaches consumers. Leaking pipes and aging infrastructure are among the main causes of this problem, wasting a limited natural resource and generating significant economic and environmental costs. Improving the monitoring of water distribution networks is therefore essential to detect abnormal conditions early and support more effective management and maintenance of these systems.

This thesis addresses these needs through the design and development of a web-based dashboard to support the activities of people involved in the management and maintenance of local water distribution systems in the province of Cuneo, Piedmont, Italy. The dashboard was designed to present network information in a clear and accessible way for operational and technical users. Requirements analysis and user-centered design were used to define its main functionalities, information structure, and interface. The interface was first prototyped in Figma and validated by professionals working in the application domain before guiding the development of the React frontend.

An AI-assisted approach based on Large Language Models (LLMs) was used during the frontend development process. In a first stage, three AI coding tools, Gemini, Codex, and Claude Code, were compared on the same implementation task, both with and without a persistent project-context file containing information about the application, its architecture, and development conventions. The comparison considered the resulting implementation and the development process. Based on this initial comparison, Gemini was then used to support the remaining implementation tasks. The subsequent development process was documented to examine the interactions and corrections required when using AI assistance across a larger frontend implementation.

The final result is a React-based web dashboard that includes interactive maps, sensor and district monitoring views, charts, filtering, and supporting management interfaces. It provides a visual and interactive way to explore information related to the monitored water distribution network. At the same time, the development process demonstrates the practical use of LLM-based coding assistance and highlights the importance of clear requirements, iterative verification, and human supervision when AI-generated code is integrated into an evolving application.

Acknowledgments

I express my sincere gratitude to my supervisor, Prof. Gianvito Urgese, for their invaluable guidance, constructive feedback, and continuous support throughout the development of this thesis. I am also grateful to Giuseppe Fanuli and Walter Gallego Gomez for technical discussions and suggestions that contributed to the design and progress of the project.

I would like to thank my family, friends, and colleagues for their encouragement and patience during the development of this work.

Contents

List of Figures 6

List of Tables 7

1 Introduction 9
- 1.1 Context and Motivation . . . 9
- 1.2 Problem Statement . . . 10
- 1.3 Project Scope . . . 10
- 1.4 Thesis Objectives . . . 10
- 1.5 Research Questions . . . 11
- 1.6 Thesis Contributions . . . 11
- 1.7 Thesis Organization . . . 12

2 Literature Review 13
- 2.1 Chapter Overview . . . 13
- 2.2 Situation Awareness and Cognitive Load in Monitoring Interfaces . 13
- 2.3 Dashboard Design, Status Communication, and Overview-to-Detail Patterns . . . 14
  - 2.3.1 Status and Alarm Design . . . 14
  - 2.3.2 Overview-to-Detail Navigation . . . 15
- 2.4 Persona-Driven and Evidence-Grounded User-Centered Design . . . 15
- 2.5 Requirements Traceability from Stakeholder Needs to Implementation 16
- 2.6 React Frontend Architecture for AI-Assisted Development . . . 16
- 2.7 AI-Assisted Coding and Repository Context Files . . . 17
- 2.8 Synthesis: Literature Gaps and Research Questions . . . 18
- 2.9 Chapter Summary . . . 18

3 Requirements Engineering and Frontend System Design 21
- 3.1 Requirements Sources and Project Context . . . 21
- 3.2 Personas and User Needs . . . 22
  - 3.2.1 Luca, Aqueduct Operator . . . 22
  - 3.2.2 Sara, Hydraulic Engineer . . . 23

3.2.3 Complementary Needs and Design Tension 23
3.3 System Requirements and Use Cases 23
3.3.1 Functional Requirements 23
3.3.2 Non-Functional Requirements 25
3.3.3 Use Cases and Scenario 25
3.4 UX and Figma Design 26
3.4.1 Status, Alerts, and Analytical Depth 27
3.4.2 Role of the Figma Prototype 27
3.5 Frontend System Design and Implementation Boundary 28
3.6 Requirements-to-Artifact Traceability 31
3.7 Chapter Summary 33

4 Methodology: AI-Assisted Design and Implementation 35
4.1 Research Design and Project Context 35
4.1.1 Target Users and Persona-Based Design 36
4.1.2 Design–Implementation Feedback Cycle 36
4.1.3 AI Roles and Instruction Layers 37
4.2 Bounded Comparative Study (RQ2a) 39
4.3 Cumulative Prototype Development (RQ2b) 48
4.4 Evidence and Analysis 49
4.5 Methodological Boundaries 49

5 Results and Discussion 51
5.1 Design and Prototype Outcome 51
5.2 What Persistent Project Context Changed 61
5.3 Observed Failure and Risk Patterns in the Cumulative Build 62
5.3.1 Scope Transfer and Recurring Defaults 63
5.3.2 Verification and Delayed Discovery 64
5.3.3 Correction Effort as a Process Signal 64
5.4 Practices Associated with Better Outcomes 65
5.5 A Case-Derived Blueprint for Project-Context Files 66
5.6 A Supervised Workflow for AI-Assisted Development 67
5.7 Implications and Limitations 68

6 Conclusions 71
6.1 What the Thesis Achieved 71
6.2 Answers to the Research Questions 71
6.3 Original Contributions 72
6.4 Future Work 73
6.5 Closing Remarks 73

A Evaluation Instrument and Evidence Archive 75
A.1 Evaluation Structure 75
A.2 Data-Completeness Note 76
A.3 Digital Archive Organization 76

B Task and Evidence Index 79

Bibliography 81

List of Figures

3.1 Navigation structure of the DigiWater frontend prototype. 26
3.2 Overview dashboard example with Marene shown as critical. 29
3.3 Implemented municipality detail dashboard using Marene as an example. 30

4.1 Iterative relationship between design, implementation, and review. . 37
4.2 Task 02.1 without project context: Codex. 40
4.3 Task 02.1 with project context: Codex. 43
4.4 Recorded total session time for Task 02.1 by assistant and context condition. 46
4.5 Recorded total context traffic for Task 02.1 by assistant and context condition, plotted on a logarithmic scale. Blue denotes the without-context condition and orange the with-context condition. 47

5.1 Login: Figma design (top) and React prototype (bottom). 53
5.2 Overview: Figma design (top) and React prototype (bottom). . . . 54
5.3 City details: Figma design (top) and React prototype (bottom). . . 55
5.4 District details: Figma design (top) and React prototype (bottom). 56
5.5 Leak reporting: Figma design (top) and React prototype (bottom). 57
5.6 Leak history: Figma design (top) and React prototype (bottom). . . 58
5.7 AI assistant: Figma design (top) and React prototype (bottom). . . 59
5.8 User profile: Figma design (top) and React prototype (bottom). . . 60
5.9 Case-derived supervised workflow for AI-assisted development. . . . 68

List of Tables

3.1 Functional requirements specified for DigiWater. 24
3.2 Requirements-to-artifact traceability chains linking persona goals to implementation state. Luca's undo need has no corresponding functional requirement or implemented mechanism and is not shown here; see the paragraphs below for full detail per chain. 32

4.1 Task 02.1 process metrics. 46
4.2 Recorded token usage for Task 02.1, excluding later retrospective report generation. 47

5.1 Task 02.1 comparison summary. 61
5.2 Case-derived taxonomy of failure and risk patterns in AI-assisted implementation. 63
5.3 Case-derived blueprint for a persistent project-context file. 66

A.1 Summary of the comparative evaluation instrument. 75
A.2 Transcript-derived prompts in the six-condition comparison. 76

B.1 Functional index of cumulative-build task material. 79

Chapter 1

Introduction

1.1 Context and Motivation

A leak that goes undetected for even a few days can waste thousands of litres of treated water and cost a utility real money and effort to trace and repair, yet in many distribution networks leaks are still found late, and mostly by chance. The scale of the problem is not local. Global freshwater demand is projected to outstrip supply by 40% by 2030, and an average of 23% of treated water in the European Union never reaches a tap, lost instead to leaking pipes and ageing infrastructure [1]. Detecting these losses earlier, and understanding where a network is behaving abnormally, is therefore not just an efficiency question but a resource one [2].

The Smart Digital Water System is a water-network monitoring project promoted by Fondazione DIG421 in collaboration with Tesisquare, Alpi Acque S.p.A., and Politecnico di Torino. Its underlying technology was developed by Politecnico di Torino and tested in several municipalities in Piedmont using operational infrastructure made available by Alpi Acque [3].

The system collects real-time flow and pressure measurements from sensors installed at key points in the distribution network and transmits them to a cloud platform. Artificial-intelligence algorithms use these data to identify leaks and anomalies, helping operators localize problems more quickly, reduce response times and maintenance costs, and make informed decisions [3].

The AI-assisted development strand is motivated by an unresolved tension between practitioner guidance and empirical research on persistent repository-level context files. The `AGENTS.md` format is presented as "a simple, open format for guiding coding agents" [4], and an analysis of over 2,500 repositories publishes concrete authoring guidance for such files [5]. By contrast, a controlled empirical study by Gloaguen et al. [6] found that providing such context files does not generally improve coding-agent task success rates, while increasing inference cost by over 20% on average. This thesis does not resolve that tension; it motivates the AI-assisted

workflow research question below.

1.2 Problem Statement

Although the sensing and AI components generate operational information, their value depends on how clearly that information is presented to water-network operators [3]. The problem addressed by this thesis is therefore the usability of the operator-facing frontend: it must support both rapid operational awareness and deeper analytical exploration – including filtering, historical analysis, drill-down, and technical detail – without overloading operational users.

This thesis addresses that gap by designing and developing the DigiWater frontend using Human–Computer Interaction (HCI) and User Experience (UX) principles. The design aims to support operator situation awareness through clear visual hierarchy, consistent status indicators, and coherent layout composition while reducing unnecessary interface-induced cognitive load. Chapter 2 situates these design choices within the relevant HCI and dashboard-design literature.

1.3 Project Scope

This thesis develops a **frontend-only** prototype of the DigiWater dashboard: a client-side web application built with React. Backend systems, authentication, databases, deployment, and real SCADA/Internet of Things (IoT) integration are explicitly out of scope; features that would normally depend on external services or backend functionality, such as authentication and conversational AI, are represented through frontend-only approximations and mock data.

The artifact implements a scoped frontend subset of the specification. Using mock data, it represents login and supports map-based leakage-risk visualization, network overview and district drill-down, pressure and flow charts, leak reporting and history, anomaly alerts, and a conversational-assistant interface. Chapter 3 presents the full functional and non-functional requirements and documents requirements that were implemented only partially or deferred.

1.4 Thesis Objectives

Building on this scope, the thesis pursues three related strands of work:

1. **Requirements engineering and UX design.** Translating the stakeholder and persona needs identified during requirements analysis into a documented set of functional and non-functional requirements, and into UX design decisions grounded in those requirements.

2. **Frontend implementation.** Delivering a working React prototype of the DigiWater dashboard that implements a scoped frontend subset of the requirements specification.

3. **Evaluation of an AI-assisted development workflow.** Assessing, through the six-condition comparison and the cumulative build under persistent project context, what the resulting record reveals about the practical role of that context in AI-assisted frontend development. Section 1.5 defines the two stages precisely; both are examined in Chapters 4–5.

1.5 Research Questions

The requirements and UX strand and the AI-assisted workflow strand define the thesis's research questions. The frontend implementation is the artifact through which both are investigated.

RQ1. How should a frontend monitoring interface for a water distribution network be designed to balance rapid operational awareness with detailed analytical exploration, while minimizing unnecessary cognitive load imposed by the interface?

RQ2. What does the implementation of DigiWater reveal about the practical role of persistent project context in AI-assisted frontend development?

RQ2a (Task 02.1 only, six conditions). For a single task implemented by three AI coding assistants (Codex, Claude, and Gemini), each with and without the fixed project-context file described above, what differences are observed in task completion, the number and purpose of build/runtime and design-fidelity follow-up prompts, the total number of user prompts, and the architectural consistency, correctness, maintainability, requirements compliance, and UX quality of the resulting code?

RQ2b (twelve-task cumulative-build subset). Across these tasks, completed with a single assistant (Gemini, via Antigravity) under that same persistent context file throughout, what does the task-by-task development record show about adherence to the file, conflicts between it and per-task instructions, requirements traceability across tasks, and correction effort, as the codebase evolved?

1.6 Thesis Contributions

Corresponding to the three objectives above, this thesis makes three overarching contributions. The third is expressed through three concrete methodological outputs:

1. **A requirements and UX design specification for the DigiWater frontend.** A documented set of functional and non-functional requirements, and

UX design decisions grounded in the Luca and Sara personas, connecting stakeholder needs to concrete interface decisions.

2. **An evolved interface design and React prototype of the DigiWater dashboard.** The revised Figma design and working frontend implementation show how a scoped subset of the specification was refined through design, implementation, visual review, and supervisor feedback, within the scope described in Section 1.3.

3. **A case-study analysis of persistent project context in AI-assisted frontend development.** Based on the six-condition comparison and the cumulative build defined in Section 1.5 (RQ2a and RQ2b), this thesis documents what was observed for this project and this context file. It is offered as a case-study-level contribution scoped to that evidence, not as a general claim about AI-assisted software development. This contribution is made concrete through:

    - a case-derived taxonomy of observed implementation failures and risks;
    - a blueprint for writing and maintaining persistent project-context files; and
    - a supervised workflow for AI-assisted development.

1.7 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 reviews related work in Human–Computer Interaction and usability design for monitoring dashboards, in requirements engineering for socio-technical systems, and in AI-assisted software development, including repository- and project-level context and persistent context/instruction files for coding agents.

Chapter 3 presents the requirements engineering process and the resulting functional and non-functional requirements, the stakeholder and persona analysis, and the UX design decisions derived from them.

Chapter 4 explains the design–implementation–review cycle and the two-stage case study used to examine persistent project context and cumulative AI-assisted development.

Chapter 5 presents the revised design and working prototype, interprets the case-study findings, and derives the failure and risk taxonomy, project-context-file blueprint, and supervised workflow.

Finally, Chapter 6 brings the three overarching contributions together, discusses their limitations, and outlines directions for future work.

Chapter 2

Literature Review

2.1 Chapter Overview

Chapter 1 introduced two analytical strands: the design of the DigiWater monitoring interface (RQ1) and the AI-assisted workflow used to implement it (RQ2). This chapter reviews the literature supporting each strand while keeping their theoretical and empirical foundations distinct.

The development context considered in this review includes Figma for interface prototyping, React for frontend implementation, and Codex, Claude, and Gemini as AI coding assistants. Figma and React support the design and implementation strand, while the assistants and their repository-level context files support the AI-assisted workflow strand. Their precise roles in the study are detailed in Chapter 4.

Sections 2.2–2.5 address RQ1 through situation awareness, cognitive load, dashboard and alarm design, overview-to-detail interaction, persona-driven design, and requirements traceability. Sections 2.6 and 2.7 address RQ2 through React project organization, AI-assisted and agentic coding, and persistent repository-level context files. Section 2.8 then relates the two strands separately to RQ1, RQ2a, and RQ2b.

Foundational theory, empirical research, industry standards, and practitioner or grey literature are interpreted according to their evidential status rather than treated as interchangeable.

2.2 Situation Awareness and Cognitive Load in Monitoring Interfaces

Situation awareness is defined by Endsley as the perception of relevant elements in an environment, the comprehension of their meaning, and the projection of their status into the near future [7]. Distinguishing these levels is useful in monitoring because perceiving a sensor value does not guarantee that a user will recognize it as

anomalous or anticipate its development. For DigiWater's operational monitoring, this corresponds to perceiving district status, interpreting anomalies, and assessing their urgency under time pressure.

Cognitive load theory provides a complementary account of how information presentation affects this process. Sweller's work shows that heavy working-memory demands can interfere with the construction and use of problem schemas [8]. The later distinction between intrinsic load, arising from the task itself, and extraneous load, arising from its presentation, identifies the latter as the aspect that interface design can most directly influence [9]. Reducing unnecessary presentation complexity can therefore support operational monitoring without removing the analytical depth, filtering, and drill-down needed for more investigative work.

Direct empirical evidence for dashboard redesign effects is more mixed than the theory alone suggests. Fang et al.'s review places interface and display design among several factors affecting situation awareness rather than identifying it as a single validated lever [10]. Cheng et al. found limited cognitive-load effects across most conditions in a dashboard designed using cognitive load theory [11], while Burnay et al. treat cognitive load as one influence on continued dashboard use [12]. Across these three studies, the pattern is consistent: presentation remains relevant, but theory-informed redesign does not by itself guarantee measurable improvements in cognitive load or situation awareness. Their settings differ from operational water-network monitoring, so they inform DigiWater's design reasoning rather than directly establishing its effectiveness.

Endsley's and Sweller's theories thus provide the rationale for situation-awareness- and cognitive-load-aware design, while the empirical literature supports a more measured expectation of its effects. The next section turns to concrete principles for status communication and analytical navigation.

2.3 Dashboard Design, Status Communication, and Overview-to-Detail Patterns

The preceding theories explain why presentation matters; this section considers two concrete design problems in DigiWater: communicating abnormal conditions and supporting movement from rapid status recognition to detailed analysis.

2.3.1 Status and Alarm Design

EEMUA Publication 191 is an industry guide to alarm system design, management, and procurement for operator-facing alarm systems [13]. Its fourth edition names alarm prioritization and the avoidance of alarm floods as dedicated topics [14], and a 2025 review of the update summarizes its emphasis on eliminating unnecessary

alarms through rationalization and reserving alarm status for conditions that genuinely require an operator response, while stressing that priority indicates only the order in which to respond and that no alarm should be ignored on that basis [15]. Although developed for industrial process control rather than web dashboards, these prioritization and flood-avoidance principles inform DigiWater's approach to distinguishing hydraulic anomalies from sensor faults without validating any particular visual implementation.

2.3.2 Overview-to-Detail Navigation

Shneiderman's visual information-seeking mantra – *overview first, zoom and filter, then details-on-demand* – describes a progression from an overview of the full dataset to focused filtering and detailed inspection [16]. In monitoring systems, this progression can support rapid recognition at the overview level while preserving filtering, drill-down, and access to underlying data for more detailed analytical work.

EEMUA 191 and the cognitive-load literature therefore guide what an overview should prioritize, while Shneiderman's taxonomy explains how users can move from that overview into progressively deeper analysis. Chapter 3 translates these principles into DigiWater's status indicators, map-based overview, and district-level detail views.

2.4 Persona-Driven and Evidence-Grounded User-Centered Design

Personas provide a way to keep design decisions focused on documented user needs rather than designer assumptions. Pruitt and Grudin emphasize that their value depends on grounding them in evidence about users, including interviews, observations, or existing requirements documentation [17]. ISO 9241-210 places the same principle within a broader human-centred design process, requiring that design be based on an explicit understanding of users, tasks, and environments, and defining context of use as the combination of users, goals and tasks, resources, and environment [18].

Taken together, these sources indicate that personas should not be treated as fictional labels detached from evidence. Their characteristics should remain traceable to documented knowledge about users, tasks, and environments, and design decisions justified through them should preserve that connection as a system evolves. Chapter 3 applies these principles in the requirements-engineering process.

2.5 Requirements Traceability from Stakeholder Needs to Implementation

Gotel and Finkelstein define requirements traceability as the ability to follow the life of a requirement forwards and backwards through the artifacts derived from it [19]. They distinguish *pre-traceability*, which connects a requirement to its origins, from *post-traceability*, which follows it through design and implementation. A recurring practical problem is that these links may exist conceptually but disappear as a project evolves because they are not made explicit in its artifacts.

For DigiWater, the relevant chain runs from stakeholder needs and personas through functional and non-functional requirements, UX decisions, task-level acceptance criteria, and implementation. Post-traceability also provides an analytical lens for selected build-task examples in which relevant guidance is distributed between project-level context and task-specific instructions. This establishes the basis for the task-level traceability analysis reported in Chapter 5.

2.6 React Frontend Architecture for AI-Assisted Development

DigiWater is a frontend-only React prototype built substantially through AI-assisted coding. Its organization is therefore relevant not only to human maintainability but also to whether successive coding tasks can extend the codebase consistently. The sources in this section are practitioner guidance and reference architectures rather than empirical studies of coding agents.

Common React guidance favors feature or domain boundaries over organization solely by file type. Wieruch recommends grouping components, hooks, and logic around the feature they support [20]; Dodds expresses the related principle of colocation, in which code remains close to where it is used until broader reuse requires otherwise [21]. The Bulletproof React reference architecture applies similar ideas at larger scale through explicit boundaries between shared infrastructure and feature modules [22].

These sources establish documented software-engineering conventions, not evidence that a particular folder structure improves AI-agent performance. They nevertheless provide a reference point for the deliberate project-organization rules encoded in DigiWater's persistent context file. Chapters 4 and 5 describe those rules and examine their role in the evolving implementation.

2.7 AI-Assisted Coding and Repository Context Files

AI-assisted and agentic software development increasingly uses persistent repository-level files to provide coding assistants with project conventions, architecture, commands, and constraints. The `AGENTS.md` community specification promotes such a file as an open convention for guiding coding agents [4]. A GitHub practitioner article, based on an analysis of more than 2,500 public `agents.md` files, derives recommendations for their content and organization [5]. These sources demonstrate practitioner adoption and recommended practice; they do not establish that context files improve task outcomes.

Gloaguen et al. provide the closest external empirical reference point. Across multiple coding agents and language models evaluated on SWE-bench tasks and repositories containing developer-authored context files, context files tended to reduce task success rates relative to providing no repository context while increasing inference cost by over 20% [6]. This controlled benchmark evidence, reported in a preprint, creates an important tension with practitioner recommendations: adoption and perceived operational usefulness do not themselves demonstrate improved task success.

Watanabe et al. examine human correction after agent-generated code across a broad sample of public GitHub repositories, not specifically React or frontend projects. They report that 83.8% of the agent-assisted pull requests studied were accepted and merged; among those, 54.9% were merged without further modification and 45.1% required additional changes, particularly for bug fixes, documentation, and adherence to project-specific standards [23]. This evidence supports treating correction effort as a process measure rather than as a general quality score for AI-generated code.

Persistent context also raises the possibility that project guidance will drift from the codebase it describes. Grabowski's Spec Growth Engine proposes coupling specifications to code and checking that relationship as both evolve [24]. The non-peer-reviewed proposal does not report an empirical evaluation and is relevant here as a formulation of the specification–code drift problem.

These sources inform DigiWater's two separate evaluation stages in different ways. Gloaguen et al. provide the closest conceptual comparison for the Task 02.1 six-condition comparison. Watanabe et al. motivate attention to correction effort, while Grabowski identifies a drift risk relevant to the separate cumulative build under persistent project context. None studies the exact combination examined here: the Task 02.1 six-condition comparison across three coding assistants on a single task and the separate cumulative build, a longitudinal single-assistant build under a fixed project-context file. The practitioner recommendations and empirical findings therefore leave the role of such context open for investigation in DigiWater.

2.8 Synthesis: Literature Gaps and Research Questions

For **RQ1** (how a frontend monitoring interface should be designed to balance rapid operational awareness with detailed analytical exploration while minimizing unnecessary cognitive load), Endsley and Sweller explain why monitoring interfaces should support situation awareness while limiting presentation-induced cognitive load. EEMUA 191 and Shneiderman provide more concrete guidance for prioritized status communication and progression from overview to filtering and details-on-demand. Pruitt and Grudin, ISO 9241-210, and Gotel and Finkelstein connect these principles to evidence-grounded personas and requirements traceability. Direct dashboard evidence is nevertheless mixed: the literature supports the design rationale more clearly than it demonstrates that redesign will produce measurable improvements. DigiWater applies these principles to a requirements and UX specification for its operational and analytical personas.

For **RQ2** broadly, and **RQ2a** specifically (the six-condition, three-tool comparison for Task 02.1), Gloaguen et al. provide the closest external empirical reference point. Their benchmark study found that repository-level context tended to reduce task success while increasing inference cost. Task 02.1 examines a narrower, project-specific comparison across three assistants and two context conditions, establishing a conceptual point of comparison rather than a replication design.

For **RQ2b** (the task-by-task record of the twelve-task analytical subset, Tasks 02.2–08.1, from the separate cumulative build under a single assistant and a persistent context file), prior work addresses individual parts of the workflow but not their combination. Watanabe et al. examine human modification of agent-generated contributions, while Grabowski formulates specification–code drift. Requirements traceability supplies a complementary lens for selected task-level examples involving persistent context, task-specific instructions, and implementation decisions. The cumulative DigiWater record is therefore examined in a longitudinal setting not directly covered by these sources.

2.9 Chapter Summary

This chapter reviewed two distinct literature strands. For RQ1, situation awareness and cognitive load theory provide the theoretical basis for reducing presentation-induced demands, while dashboard evidence shows that measurable redesign effects remain mixed. Alarm-design guidance, overview-to-detail interaction, personas, and traceability provide the more concrete principles used in DigiWater's requirements and UX process.

For RQ2, React organization offers practitioner conventions for maintaining a predictable codebase, while the context-file literature reveals a tension between

practitioner recommendations and controlled evidence on task success. Research on correction effort and specification drift further informs the cumulative-build analysis.

The next chapter presents the requirements engineering process for DigiWater, the resulting functional and non-functional requirements, and the UX design decisions derived from the project's operational personas and from the dashboard-design and traceability principles reviewed above.

Chapter 3

Requirements Engineering and Frontend System Design

3.1 Requirements Sources and Project Context

This chapter explains how DigiWater's project requirements and user needs were translated into UX, interface, and frontend implementation decisions. The requirements presented here were derived through a requirements engineering process conducted as part of this thesis. Its principal source is the *DigiWater Engineering Requirements Document*, which defines the stakeholders, personas, functional and non-functional requirements, use cases, and an intended high-level system design. The Figma prototype and the implemented React application provide the subsequent design and artifact evidence. No interviews, observations, or other primary user research were conducted as part of this thesis.

These sources describe different stages and must therefore be kept distinct. Statements that the specification *requires* or *envisages* refer to the intended DigiWater system. Statements that the thesis prototype *implements* or *represents* refer to the frontend-only artifact. In particular, the project documentation assumes an available backend and external data providers, whereas the prototype uses local mock or hardcoded data and does not implement backend integration, authentication, database persistence, deployment, or real SCADA/IoT ingestion.

The public project description identifies real-time flow and pressure sensing, cloud data transmission, and AI-assisted leak and anomaly detection as the system's operational context. It also emphasizes helping operators localize leaks more quickly and make informed decisions through an accessible dashboard [3]. For this thesis, these capabilities motivate the frontend requirement to present sensor, prediction, and network information through an integrated interface in which important conditions can be recognized at a glance.

The actors relevant to the requirements are described by role rather than by treating the organizations involved in development as target stakeholders. The

application is intended for a generic aqueduct company, whose control-room operators and hydraulic engineers are the principal users. In the DigiWater project, Alpi Acque represents this operational organization and supported real-world testing by providing access to its infrastructure [3]. The other participating organizations supplied project coordination, funding, research, design, and implementation support and are therefore treated as the development team rather than as target stakeholders. External technical providers supply SCADA telemetry and geospatial topology, but their systems remain outside the thesis prototype. The operational roles are represented by the Luca and Sara personas discussed below.

Some requirements are not tied to either persona. The personas primarily organize requirements for monitoring and technical analysis; entry-point and system-wide requirements, such as the login page and alerts and anomaly awareness, have a broader project provenance. This distinction prevents entry-point or cross-cutting functionality from being assigned artificially to Luca or Sara.

3.2 Personas and User Needs

The requirements document defines two personas as design abstractions for operational users and uses them to guide UI/UX decisions and requirement prioritization. As discussed in Section 2.4, their evidential basis is the existing project documentation rather than primary user research.

3.2.1 Luca, Aqueduct Operator

Luca is a front-line technician who monitors the network from an aqueduct company's control room, including during night and emergency shifts. He has high proficiency in hydraulic infrastructure but low–medium proficiency with software dashboards, and often works under time pressure. The source scenario describes a sudden pressure drop at 03:00, when he must quickly distinguish a pipe burst, pump failure, or sensor anomaly and decide whether to dispatch a field crew.

His goals are to understand district status at a glance, localize anomalies rapidly, and navigate without technical configuration or query languages. His frustrations are the need to correlate information across fragmented tools and the cognitive overload caused by excessive alerts or overly detailed views during critical situations. The requirements document therefore identifies three critical needs: high-contrast green/red status indicators, prioritization of critical alerts to prevent alarm fatigue, and clear confirmation and undo mechanisms for interaction under stress.

3.2.2 Sara, Hydraulic Engineer

Sara performs office-based analytical work and has high proficiency in hydraulics and data analysis. She validates system behavior and investigates anomalies using historical and real-time data. Her goals include comparing observations with simulations or reference behavior, correlating pressure and flow to distinguish leaks from sensor faults, and exploring hypothetical network responses.

Her principal frustrations are alerts that do not expose their underlying data or logic and missing or incomplete measurements that restrict analysis. The requirements document associates these needs with multi-variable time-series visualizations, optional access to calculation details, and flexible filtering and comparison tools.

3.2.3 Complementary Needs and Design Tension

Luca's need for rapid recognition favors concise, high-contrast, alarm-prioritized views. Sara's work requires analytical depth, transparency, filtering, and comparison. The intended interface therefore has to support both quick operational awareness and progressive access to detail rather than treating one mode as universally sufficient. This tension motivates the overview-to-detail structure developed in Section 3.4. It does not imply that every need was realized equally or completely in the thesis prototype.

3.3 System Requirements and Use Cases

3.3.1 Functional Requirements

Table 3.1 preserves the intended functional requirements defined in the project specification. Their implementation status is addressed separately in Sections 3.5 and 3.6; partial or deferred implementation does not alter the original requirement.

Table 3.1: Functional requirements specified for DigiWater.

ID	**Primary Persona(s)**	**Specified requirement**
FR0	–	Login Page: entry point where DigiWater's operational users authenticate before accessing the system; not a public or unauthenticated page.
FR1	Luca	District Visualization: display each district with a color representing its leakage probability so that operators can identify critical areas rapidly.
FR2	Luca; Sara	District Drill-Down: selecting a district opens a detailed view with synchronized charts, status indicators, and contextual explanations for rapid response and in-depth analysis.
FR3	Sara	Chart Display: render pressure and flow time-series charts, supporting comparison across different time ranges for anomaly investigation and validation.
FR4	Luca	Maintenance Form: provide a validated form for submitting repair entries and maintaining consistency and traceability after field interventions.
FR5	Luca; Sara	Leak History & Filtering: display a filterable, paginated history of reported leaks and maintenance interventions, supporting historical analysis and traceability of past field actions.
FR6	–	Alerts and Anomaly Awareness: highlight active anomalies and distinguish hydraulic events from sensor malfunctions so that the appropriate response team can be identified.
FR7	Luca; Sara	System Overview Metrics: provide aggregated indicators, including active anomalies, resolved events, and overall network status, for situational awareness and technical validation rather than management or regulatory reporting.
FR8	Luca	Conversational Assistant UI: provide a persistent natural-language interface for system-status queries or navigation to districts, consuming the intended backend AI service.

FR0 is the login entry point and lies outside the two operational-persona workflows. FR6 is a cross-cutting system requirement for which the source does not identify a primary persona. Leaving this entry unassigned preserves the source distinction rather than forcing all system functionality into a persona mapping.

3.3.2 Non-Functional Requirements

The specification also defines five non-functional requirements, motivated at category level by Luca's stress conditions and Sara's need for analytical depth:

- **Usability:** the interface should be understandable with minimal training and support task-oriented workflows.
- **Performance:** it should operate smoothly on standard office computers and provide timely visual feedback after filters or selections change.
- **Reliability:** it should handle slow or missing data gracefully and communicate data freshness and availability.
- **Scalability:** it should accommodate additional districts without major redesign, using consistent configuration patterns.
- **Portability:** it should run on major desktop browsers.

These requirements served as design targets. This chapter does not treat them as experimentally demonstrated properties: no controlled usability, performance, reliability, scalability, or cross-browser study was conducted for the prototype.

3.3.3 Use Cases and Scenario

The requirements document maps four high-level use cases to the functional requirements:

- **UC1 – Monitor District Status (Luca):** review the map to identify high-risk districts (FR1, FR2).
- **UC2 – Analyze Anomaly (Sara):** compare real-time data with simulations to validate a leak (FR3, FR7).
- **UC3 – Record Field Action (Luca):** log a maintenance intervention after repairing a leak (FR4).
- **UC4 – Ask Digital Assistant (Luca):** ask for a specific district's current pressure (FR8).

The accompanying intended scenario, *Luca's Morning Network Check*, combines several of these requirements through a concrete example. Marene is a municipality in Piedmont used as one of the project's pilot locations; it serves here only as an example aqueduct area rather than as the sole target of the application [3]. Luca logs in through the specified entry page and sees Marene highlighted in red, where

red denotes a critical leakage status requiring immediate attention. He opens its flow and pressure charts, asks whether the pressure drop may be a sensor fault, receives a likely-leak response, dispatches a crew, and later records the intervention. This scenario specifies the desired end-to-end interaction and is illustrated by the overview and drill-down examples in Figures 3.2 and 3.3. In the thesis artifact, the login, map, drill-down, form, and chat surfaces are represented at frontend level, but the authentication, live diagnosis, backend response, and persistent record are not implemented.

3.4 UX and Figma Design

The requirements document proposes visual hierarchy and layout composition as means of making critical conditions recognizable at a glance. The design translates that intention into the information architecture shown in Figure 3.1. After the login entry point, a persistent sidebar provides access to the overview, aqueduct hierarchy, leak-report form, leak history, and settings. Expanding the aqueduct branch leads first to a municipality and then to its district-level detail, while the AI chat panel is available as an overlay throughout the dashboard. This structure follows Shneiderman's progression from overview to filtering and details-on-demand [16]: the overview emphasizes operational status, while deeper views expose more technical information.

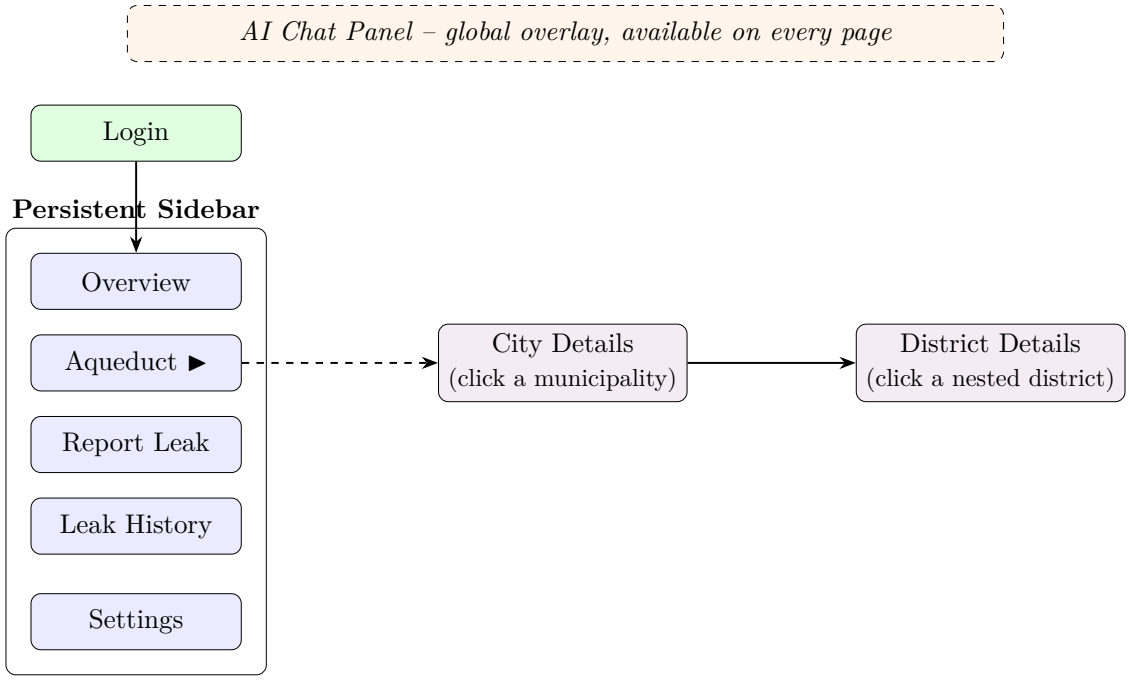


Figure 3.1: Navigation structure of the DigiWater frontend prototype.

3.4.1 Status, Alerts, and Analytical Depth

For Luca, the overview gives visual priority to district condition through high-contrast status colors and compact status summaries. The original requirement specifies a binary green/red treatment; the final interface design extends this to normal, warning, and critical levels so that an intermediate condition does not have to be presented as either fully normal or fully critical. Its asymmetric map-and-issues layout gives the map more space than the accompanying table, reinforcing the map as the primary overview surface. Alert presentation also distinguishes leakage warnings from sensor warnings, realizing FR6's required separation between hydraulic events and sensor malfunctions. These choices are consistent with the status-prioritization and alarm-fatigue guidance discussed in Section 2.3 [13, 15].

For Sara, the city and district pages provide drill-down, pressure and flow charts, status history areas, sensor summaries, and leak-history access. These design elements support movement toward analytical detail, but the prototype does not provide the full flexibility envisaged by Sara's needs: calculation logic, genuine simulation comparison, hypothetical analysis, and flexible time-range comparison remain absent or only visually represented.

The maintenance form applies required-field checks and provides a visible success confirmation after submission. This makes the submission outcome explicit, though its effect on user uncertainty was not evaluated. This supports the confirmation part of Luca's stated need, but it does not provide an undo mechanism and should not be treated as satisfying that separate need.

3.4.2 Role of the Figma Prototype

Figma provided the visual reference that connected the high-level requirements to concrete pages and interface regions. Its frames covered the login, overview, city details, district details, leak-report form, leak-history table, assistant panel, and user-profile view. Collectively, they established the page hierarchy, the persistent sidebar and top-bar shell, the relative prominence of overview components, and recurring patterns for cards, tables, status pills, forms, and detail panels. The Figma material was therefore a design input to the frontend build, not an experimental result or evidence that the design improved user performance.

The designs were not followed mechanically when a documented usability consideration supported refinement. The KPI cards, for example, were increased from the literal Figma typography values of 11px, 30px, and 10px to 13px, 32px, and 12px to improve status readability for the overview's operator-facing use, a persona-informed refinement discussed further in Section 5.4. Its effect on actual users was not empirically tested. By contrast, choices such as data structures, routing, status-computation helpers, and mock-data handling are technical implementation decisions and are addressed in the next section.

3.5 Frontend System Design and Implementation Boundary

The project specification envisages a React client divided conceptually into login/entry, dashboard, and interaction modules. It further envisages a Leaflet map receiving backend status data, pressure and flow visualization through Grafana or Chart.js, a persistent chat interface connected to a backend conversational service, and representational state transfer (REST) communication with the wider DigiWater infrastructure. These statements define the intended system architecture, not the implemented scope of this thesis.

The thesis artifact is a client-side React and TypeScript prototype built with Vite. React Router provides separate routes for the login, overview, city, district, report, history, and profile/settings views. A shared dashboard layout supplies the sidebar, top bar, and assistant shell, while reusable cards, tables, charts, status components, and map components keep presentation patterns consistent across pages. Plain CSS and shared style conventions implement the Figma-derived visual language.

Figure 3.2 provides a concrete example of the implemented dashboard. The top cards summarize network-wide conditions, the map locates the monitored aqueduct areas, and the issues table identifies the municipality, issue type, district, and severity requiring attention. In this example, Marene is marked red and listed as *Critical*, demonstrating how the overview combines visual prioritization with explicit textual status.

After selecting a municipality from the overview, the operator reaches the city-level page illustrated in Figure 3.3. Marene is used here as a concrete example of the generic drill-down design. The page identifies the municipality and its current status, displays its network on a map, summarizes recent detected-loss history, and lists the constituent districts with sensor counts, leakage probabilities, status labels, and update times. For example, the table makes the high leakage probability and critical status of the Marconi district visible without making Marene itself the sole target of the application.

Leaflet provides the interactive overview map. The implementation example uses four municipality-boundary coordinate sets identified in the archived implementation as ISTAT-derived data obtained through the openpolis `geojson-italy` mirror, rather than the earlier point-based area approximations. Within the Marene example, the named operational districts are not official administrative units with available boundary polygons, so they appear only in the navigation and detail structure, not as separate map areas. Status values still come from local mock data, not a backend or real sensor feed. The frontend helper `getEffectiveSeverity` propagates the most severe district status to its parent municipality for display.

All operational content in the prototype is local or simulated. District, sensor,

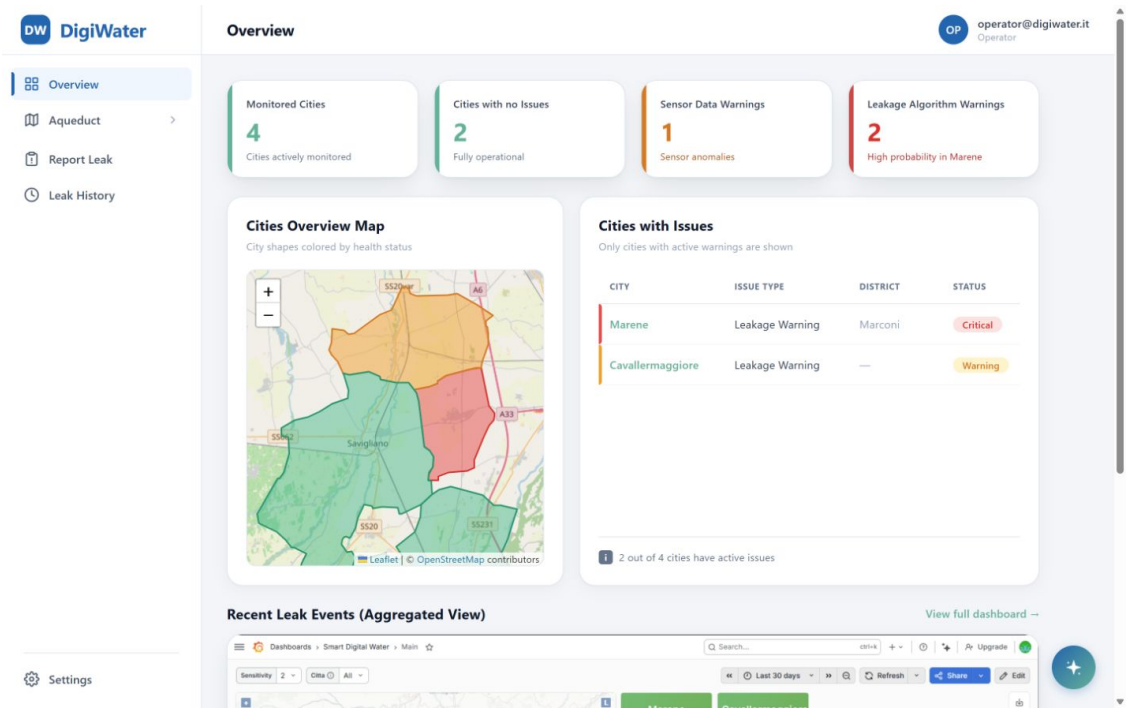


Figure 3.2: Overview dashboard example with Marene shown as critical.

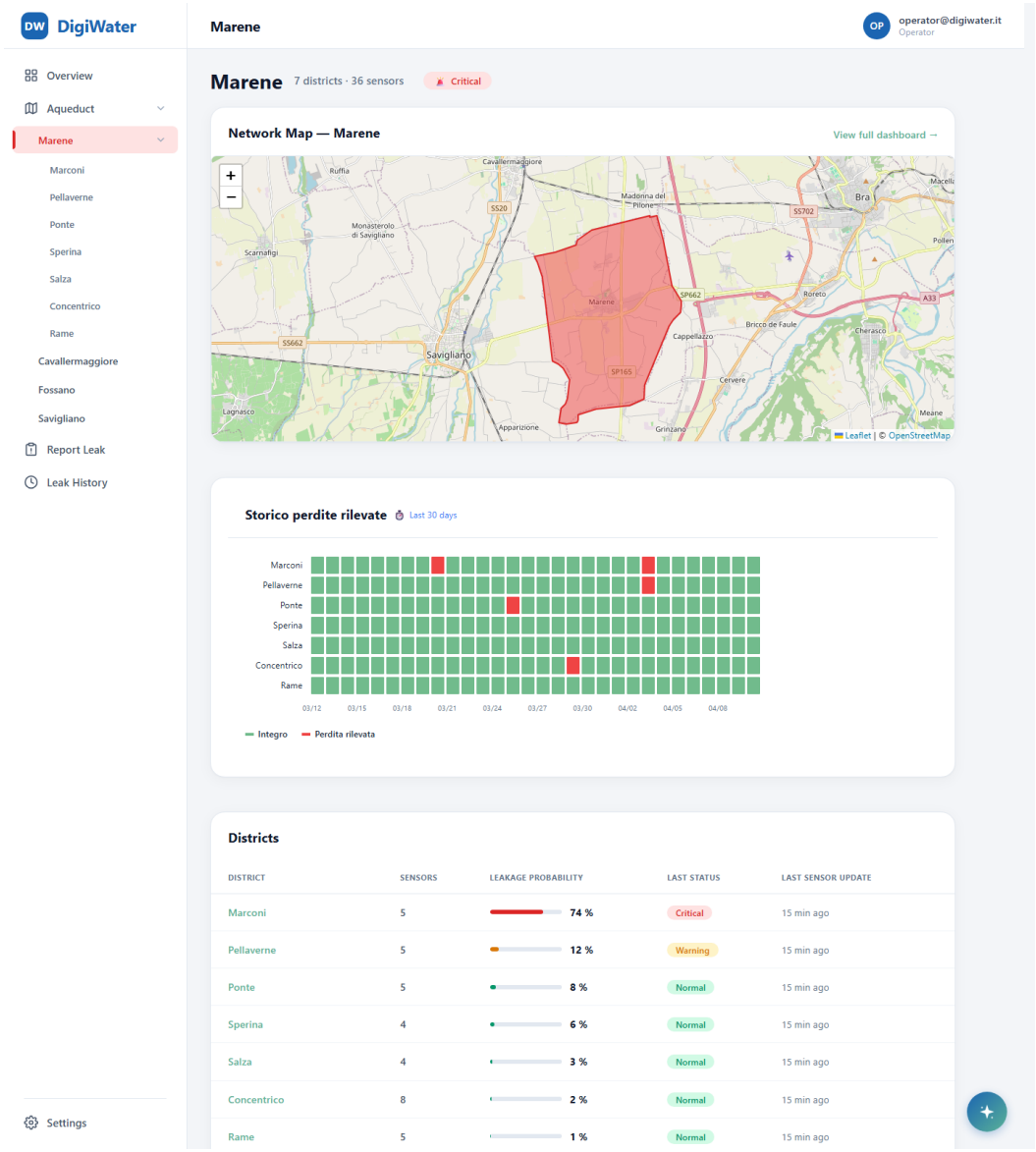


Figure 3.3: Implemented municipality detail dashboard using Marene as an example.

alert, user, and leak-report records are held in frontend mock data; sensor time series are generated locally; and Grafana content is represented by static images or explicit unavailable states. Form submission changes an in-memory array for the current session rather than writing to a database. Consistent with this thesis's frontend-only scope (Section 1.3), the login screen represents the login step visually but does not perform real authentication: no backend exists to verify credentials against, so it forwards directly to the overview. Profile changes are not persisted, and the assistant accepts text but returns a canned prototype response rather than calling an AI service.

Consequently, several specified capabilities remain outside the artifact: FR0's login page exists as a UI screen but does not perform real authentication because there is no backend against which to verify credentials; FR8's backend conversational behavior is represented only by its interface; and the backend-dependent parts of FR2, FR3, FR4, FR6, and FR7 operate only through mock data, generated values, or static placeholders. This boundary preserves the requirements as statements of intended functionality while preventing the frontend representation from being described as production integration.

3.6 Requirements-to-Artifact Traceability

These are representative traceability chains, not an exhaustive mapping of every need to every implementation detail.

Partial chain – geographic anomaly awareness (Table 3.2, row 1). The map shows overview status and anomaly priority, but not every locality as its own colored polygon, and its effect on recognition speed was not user-tested.

Partial chain – analytical investigation and model validation (Table 3.2, row 2). The charts use mock data with a fixed time window; calculation logic, simulations, and flexible filtering are not implemented.

Deferred backend-dependent chain – conversational assistance (Table 3.2, row 3). The chat panel's UI is built, but it does not evaluate queries, navigate to districts, or call a real AI service – that requires a backend outside this thesis's scope.

Implemented chain – leak history and traceability (Table 3.2, row 4). Filtering and pagination both work, supporting Luca's and Sara's needs directly, though against local mock data rather than a live backend.

A few links fall outside the table: FR0's login screen exists visually but performs no real authentication, and Luca's explicit undo need has no implemented mechanism at all. These gaps are part of the record, not exceptions hidden from it.

Table 3.2: Requirements-to-artifact traceability chains linking persona goals to implementation state. Luca's undo need has no corresponding functional requirement or implemented mechanism and is not shown here; see the paragraphs below for full detail per chain.

Persona Goal	**Requirement**	**Design Artifact**	**Implementation**	**Status**
Luca: assess all districts at a glance	FR1: color-coded district visualization; Leaflet map, colored status	Figma overview: map + status summaries prominent	Interactive map, 4 municipality polygons, normal/warning/critical from mock data	**Partial**
Sara: correlate pressure/flow, compare with simulations, flexible analysis	FR2, FR3: drill-down, pressure/flow time-series comparison	Figma district view	React drill-down, sensor charts, real-vs-predicted comparison	**Partial**
Luca: persistent NL interface, query district pressure (UC4)	FR8	Figma assistant-panel treatment + Task 07.1 reference mockups	Open/close, message-entry, message-display implemented	**Deferred**
Luca: trace past field actions; Sara: correlate leak history for analysis	FR5: filterable, paginated leak/intervention history	Figma Leak History view (filter bar + table + pagination)	React filter bar, table, pagination (Task 06.1)	**Implemented**

3.7 Chapter Summary

This chapter distinguished the intended DigiWater system from the frontend-only thesis artifact. The project documentation supplied the stakeholder context, the Luca and Sara personas, nine functional requirements, five non-functional design targets, and four operational use cases. Luca's need for rapid status recognition and Sara's need for deeper analysis motivated an overview-to-detail information architecture, while Figma translated that structure into concrete pages and recurring interface patterns.

The implemented React prototype realizes the principal navigation, overview, drill-down, chart, form, history, profile, alert, and assistant surfaces using local mock or generated data. It does not implement the specified backend integrations, production authentication, persistence, real telemetry ingestion, Grafana connections, or conversational AI service. The representative traceability examples consequently include an implemented frontend chain, two partial chains, and a deferred backend-dependent chain. Chapter 4 next describes the AI-assisted workflow used to construct and evaluate this artifact.

Chapter 4

Methodology: AI-Assisted Design and Implementation

This chapter explains how the DigiWater interface was designed, implemented, reviewed, and studied. The work combined two connected strands. The first translated requirements and personas into an interface design and a working frontend prototype. The second used that implementation process as a case study of AI-assisted development, with particular attention to persistent project context, task-specific instructions, human review, and correction.

The methodology was organized around a simple idea: neither the design nor the implementation was treated as final at the beginning. Requirements and personas informed an initial Figma design; that design guided implementation; and issues revealed during coding, visual review, and supervisor feedback informed later design and process decisions. The resulting evidence was used to answer the research questions and to derive practical guidance for future AI-assisted projects.

4.1 Research Design and Project Context

The study used a comparative stage and a cumulative stage. The comparative stage examined one bounded implementation task under six conditions: Codex, Claude, and Gemini each worked once with and once without the same persistent project-context file. This stage addressed RQ2a by holding the starting code, task specification, and design reference constant while varying the assistant and the presence of persistent context.

The cumulative stage followed the development of the broader DigiWater prototype. It addressed RQ2b by observing how instructions, implementation precedent, review, and correction interacted as the codebase grew. This was not a second controlled experiment: the cumulative build used Gemini with a persistent context file throughout. It is therefore interpreted as a longitudinal project case rather than as

causal evidence of the file's effect.

4.1.1 Target Users and Persona-Based Design

Two personas developed during requirements analysis anchored the interface requirements and design decisions. Chapter 3 documents their derivation and requirements; they are summarized here to explain how the user analysis informed the design process.

- **Luca, Aqueduct Operator.** A front-line technician monitoring the network from an aqueduct company's control room, including night and emergency shifts. He has extensive field and hydraulics experience but low–medium familiarity with software dashboards, and often works under time pressure. He needs an immediate, at-a-glance status assessment of all districts and fast localization of anomalies, without query languages or technical configuration.

- **Sara, Hydraulic Engineer.** An office-based analyst with high technical proficiency who validates system behavior and investigates anomalies using historical and real-time data. She needs multi-variable time-series comparisons, drill-down into underlying data, and flexible filtering to distinguish true leaks from sensor faults.

Together, these personas defined a central design challenge: Luca's scenarios called for high-contrast status indicators and alarm prioritization to limit cognitive overload under stress, while Sara's required analytical depth and transparency into the data behind an alert. The interface was designed to accommodate both sets of needs.

4.1.2 Design–Implementation Feedback Cycle

Chapter 3 established the requirements, personas, and initial UX rationale. These informed the first Figma designs, which then served as the main visual reference during implementation. The relationship was not one-directional. Building and reviewing the interface exposed ambiguities, impractical details, missing states, and opportunities to improve clarity. Supervisor feedback provided an additional source of design review. Where appropriate, these observations were incorporated into revised Figma screens and into later implementation instructions.

As shown in Figure 4.1, review fed back into both the design and the instructions used for subsequent implementation work.

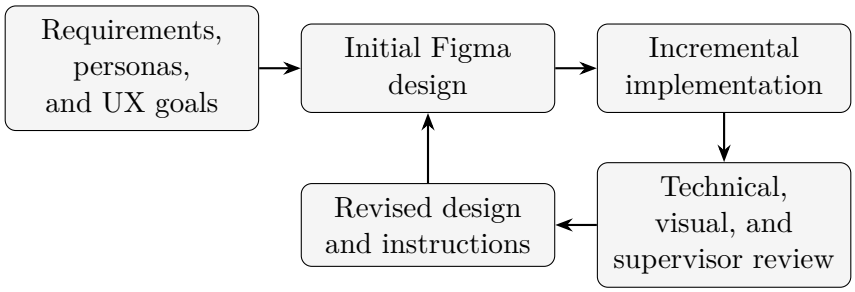


Figure 4.1: Iterative relationship between design, implementation, and review.

This cycle matters when interpreting the results. The final Figma exports are design artifacts produced through this iterative process; they are not screenshots of the running application. Conversely, the source code, build records, interaction checks, and application screenshots provide evidence of implementation. Chapter 5 reports these outcomes separately.

4.1.3 AI Roles and Instruction Layers

Codex, Claude, and Gemini were the implementation assistants in the bounded comparison. Gemini, accessed through the Antigravity IDE, implemented the cumulative build. From the middle of that build onward, Claude also supported review: it inspected prompts, plans, code, and visual evidence, helped identify root causes, and assisted in formulating corrections. The researcher retained responsibility for scope, design interpretation, acceptance, and the final decision to revise or retain an implementation.

The assistants received information through several layers. Stable project facts were stored in a repository-level context file, named `AGENTS.md` for Codex and Claude and `GEMINI.md` for Gemini. The text was otherwise the same. It described the frontend-only scope, the Luca and Sara personas, the technical stack, repository structure, coding conventions, and broad UX principles. The file was intentionally stable and did not contain task progress.

The following excerpts reproduce representative parts of the persistent context file. They are separated here so that the scope, persona, and implementation layers can be examined individually rather than compressed into one example.

Listing 4.1: Project scope excerpt.

```
Audience: technical operators, not consumers -- prioritize clarity
and fast status recognition over decoration.

Backend, auth, databases, deployment, and real SCADA/IoT
integration are out of scope.

Where a prompt and the documented requirements conflict, follow the
documented requirements.
```

Listing 4.1 establishes who the interface serves, excludes backend and live infrastructure work, and tells the assistant how to resolve a conflict between a task prompt and the requirements specification.

Listing 4.2: Persona-guidance excerpt.

```
Luca -- Aqueduct Operator. [...] Works under time pressure. Needs:
instant at-a-glance status, fast anomaly localization, no query
languages.

Sara -- Hydraulic Engineer. [...] Needs: compare observed vs.
expected data, correlate pressure/flow to rule out sensor faults,
explore history/scenarios.

Default to Luca's needs on overview/status surfaces, Sara's on
detail/analysis surfaces.
```

Listing 4.2 converts the persona analysis into a stable design rule: operational overview surfaces prioritize rapid recognition, whereas analytical detail surfaces can expose denser comparison tools.

Listing 4.3: Implementation-guidance excerpt.

```
Map: leaflet + react-leaflet + OpenStreetMap -- already installed,
use these; don't swap for Mapbox/Google Maps.

Mock data only, no real network calls. No backend/DB/auth provider.

Functional components + hooks only, no classes.
Reuse existing components before adding new ones.
Touch the minimum files needed; preserve existing conventions;
no unrelated refactors.
```

Listing 4.3 shows that the file also constrained technology choices and change discipline. These rules made outputs easier to compare because assistants were expected to use the same mapping stack, mock data boundary, React style, and repository conventions. By contrast, the current task prompt specified the feature to build, the attached design frame, and its acceptance conditions.

Each implementation step also had a task-specific prompt and, where relevant, a Figma reference. These sources served different purposes: persistent context

defined long-lived boundaries, while the current prompt defined the immediate goal and acceptance conditions. One question examined by the study was what happened when these layers aligned, omitted a needed rule, or pointed in different directions.

4.2 Bounded Comparative Study (RQ2a)

The comparative stage used the navigation task identified in the project archive as Task 02.1. Each assistant received the same frozen baseline, specification, design frame, and supporting material. For each assistant, one run included the project-context file and one did not. Separate repositories prevented one condition from inheriting code or conversation history from another.

Every run began with the same initial request. Follow-up prompts were permitted only when a build or runtime failure prevented evaluation, or when a visible design discrepancy required correction. This distinction allowed the study to separate basic implementation effort from later design-fidelity effort.

The six outputs were evaluated using the same procedure. The application was built and linted, opened at a consistent desktop viewport, exercised through the relevant interactions, and visually compared with the design reference. The evaluation considered requirements compliance, code health, visual and UX fidelity, completion, elapsed time, and the number and purpose of follow-up prompts. These ratings are descriptive researcher judgments rather than standardized usability measurements.

The frozen prompts, transcripts, repositories, screenshots, and evaluation workbook form the evidence base for this stage. Appendix A summarizes the evaluation instrument and Appendix B indexes the task materials retained in the digital archive.

Figures 4.2 and 4.3 show the running interfaces retained from the six conditions. These are application screenshots, not Figma frames. They make visible both the common result—a persistent sidebar and synchronized top bar—and implementation differences such as the depth of the expanded Aqueduct hierarchy, the selected navigation item, and the amount and treatment of placeholder content.

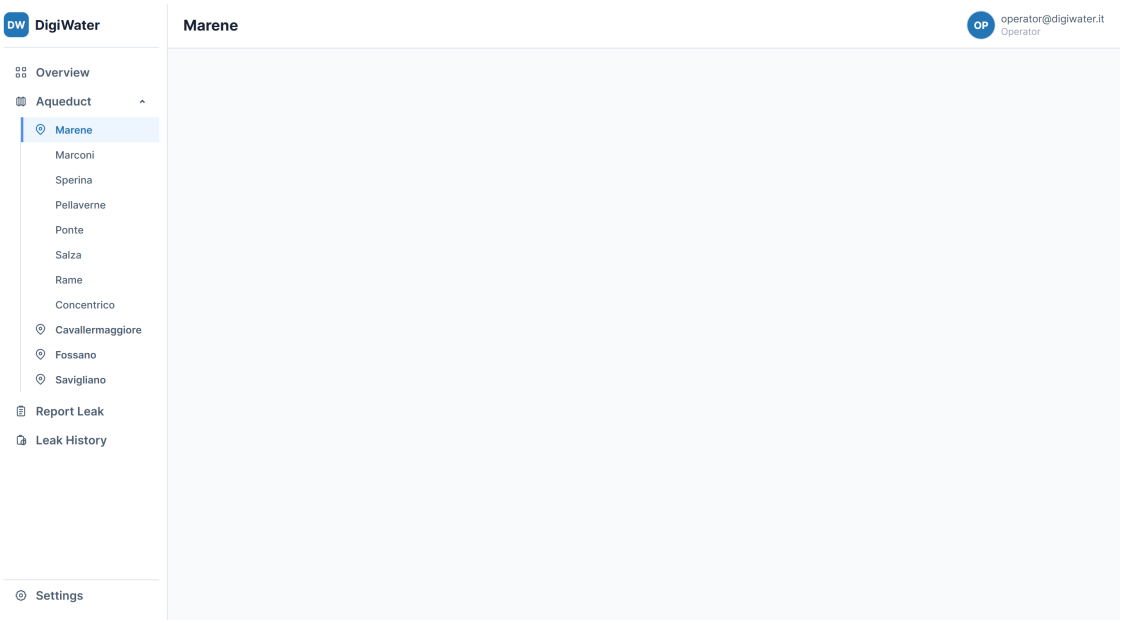


Figure 4.2: Task 02.1 without project context: Codex.

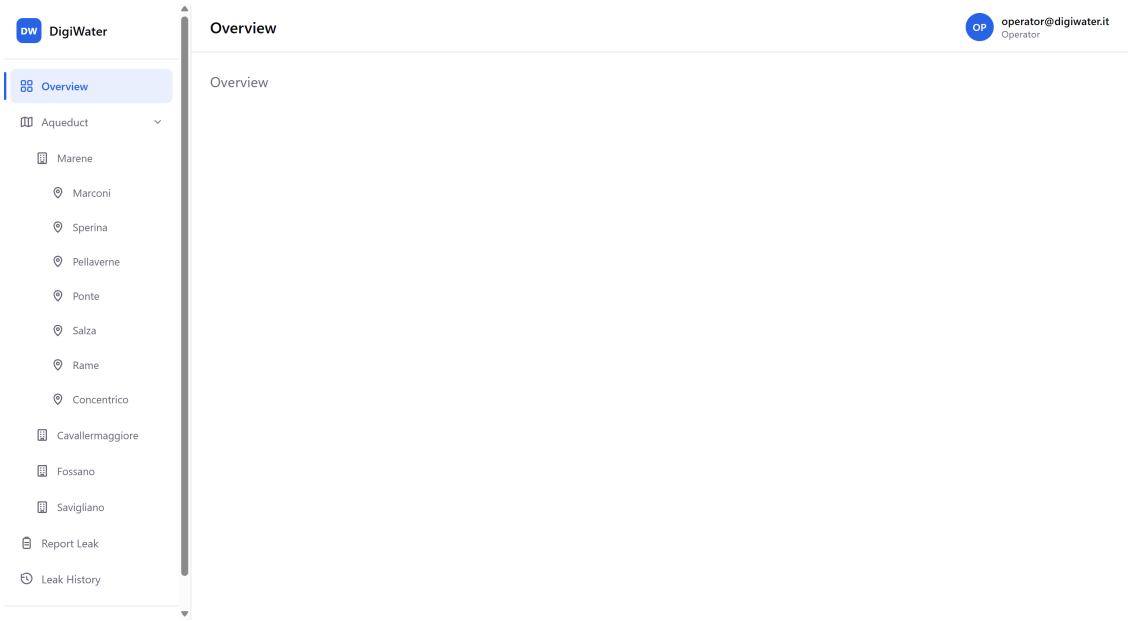


Figure 4.2: Task 02.1 without project context: Claude.

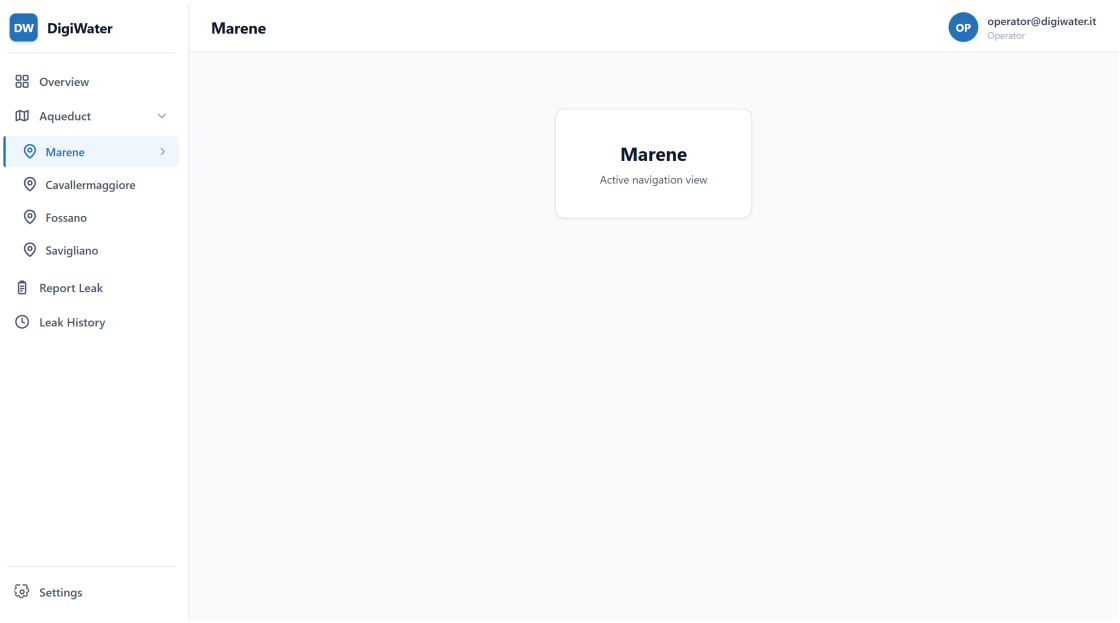


Figure 4.2: Task 02.1 without project context: Gemini.

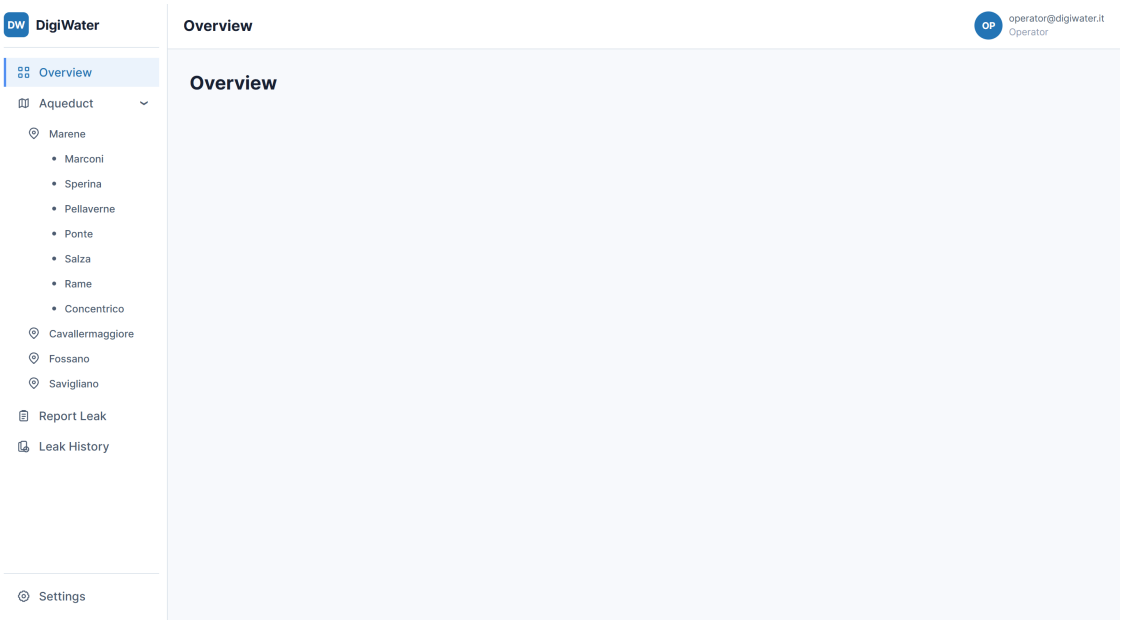


Figure 4.3: Task 02.1 with project context: Codex.

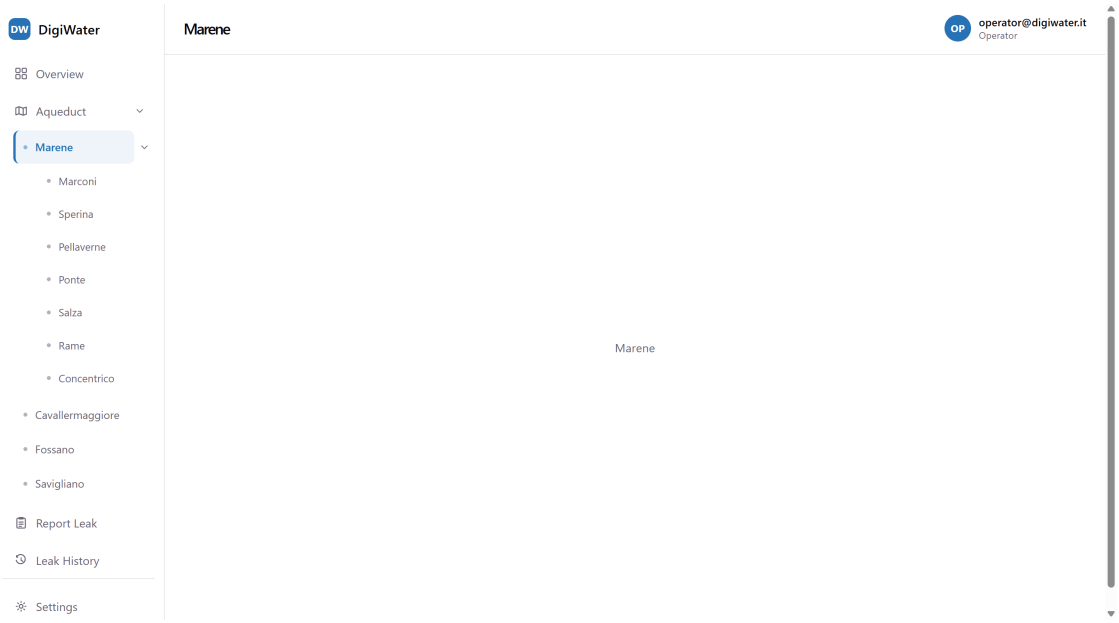


Figure 4.3: Task 02.1 with project context: Claude.

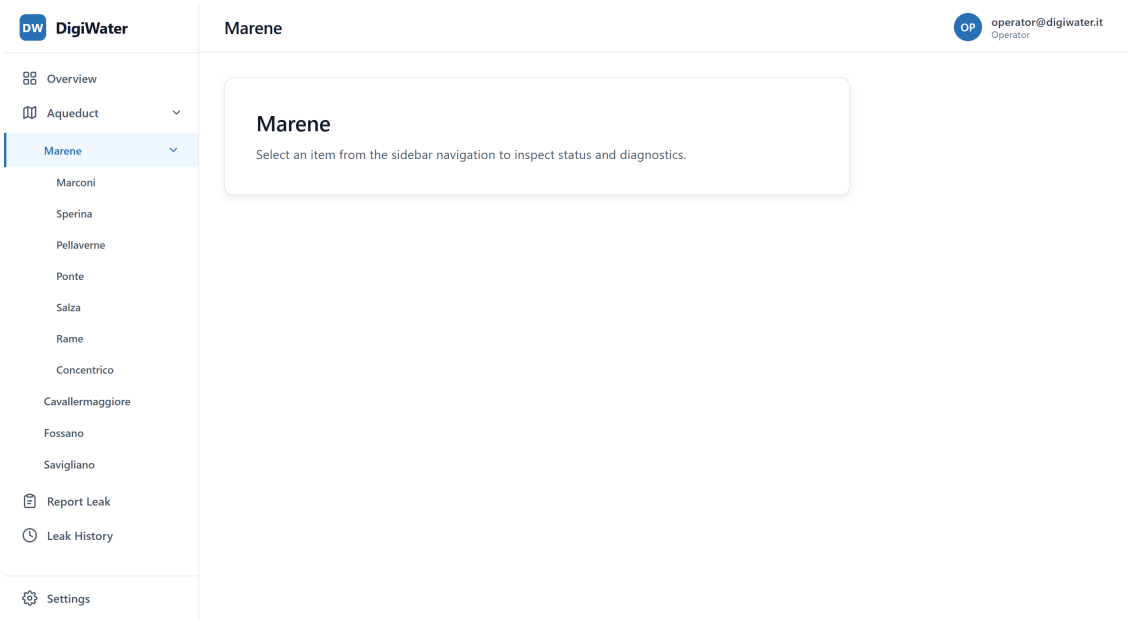


Figure 4.3: Task 02.1 with project context: Gemini.

Table 4.1 reports the two process metrics recorded for the six runs: total session time and total user prompts. Total session time includes generation, verification, and any design-fidelity correction, so it represents the practical effort required to reach the retained result rather than model latency alone. The prompt count includes the shared initial request and any permitted follow-up prompts needed to reach that result. The recorded times were 2.60 and 2.83 minutes for Codex, 8.83 and 24.50 minutes for Claude, and 8.22 and 9.00 minutes for Gemini; the corresponding prompt counts were two and two, two and three, and one and three, for the without- and with-context conditions, respectively.

Table 4.1: Task 02.1 process metrics.

Assistant	**Context**	**Time (min)**	**User prompts**
Codex	Without	2.60	2
Codex	With	2.83	2
Claude	Without	8.83	2
Claude	With	24.50	3
Gemini	Without	8.22	1
Gemini	With	9.00	3

Figure 4.4 visualizes the total-time column of Table 4.1.

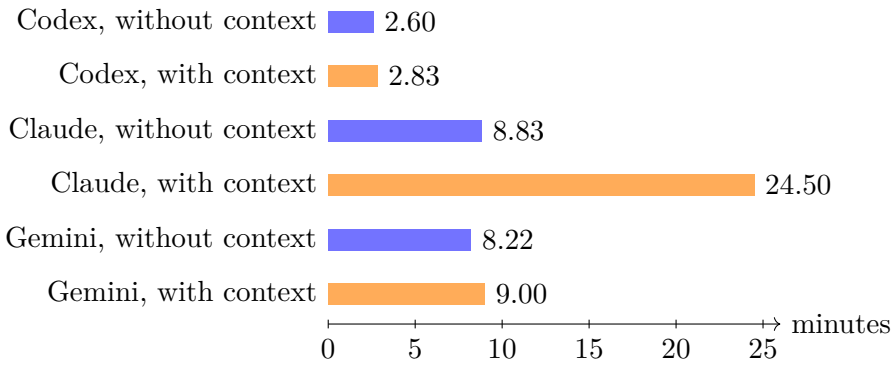


Figure 4.4: Recorded total session time for Task 02.1 by assistant and context condition.

Token use was reconstructed from the assistants' retained machine-readable session records rather than estimated from visible transcript text. To align the different logging formats, total context traffic was defined as uncached input plus cache-related input plus output. For Codex, cached input was logged as a subset of total input and was subtracted to obtain the uncached value. For Claude, cache creation and cache reads were combined in the cache-input column; Gemini recorded

cache reads separately. Activity beginning with the later request to "write a report" was excluded from every run so that the values in Table 4.2 describe only the implementation experiment.

Table 4.2: Recorded token usage for Task 02.1, excluding later retrospective report generation.

Assistant	Context	Uncached input	Cache input	Output	Total traffic
Codex	Without	38,876	289,024	5,160	333,060
Codex	With	40,348	315,648	5,465	361,461
Claude	Without	2,022	3,001,716	31,392	3,035,130
Claude	With	233	13,893,092	81,925	13,975,250
Gemini	Without	284,314	2,058,386	52,505	2,395,205
Gemini	With	490,097	1,381,200	50,769	1,922,066

Figure 4.5 visualizes the total-traffic column of Table 4.2. A logarithmic horizontal axis is used because the observed totals span more than one order of magnitude; the labels retain the exact recorded values.

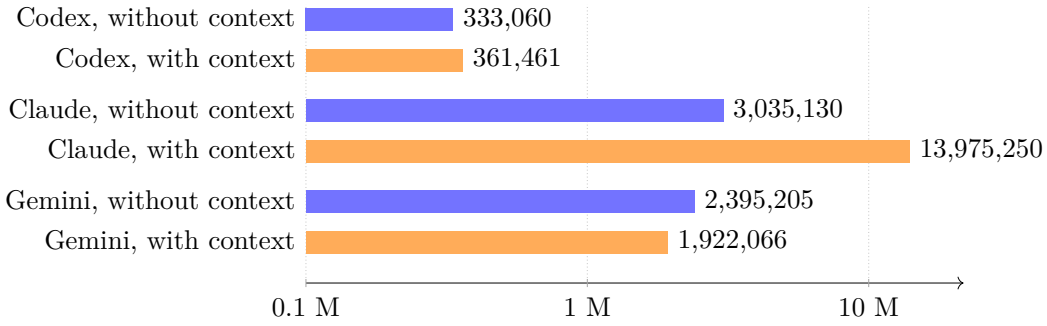


Figure 4.5: Recorded total context traffic for Task 02.1 by assistant and context condition, plotted on a logarithmic scale. Blue denotes the without-context condition and orange the with-context condition.

The Gemini records further divide output into reasoning and response/tool tokens: 30,193 and 22,312 tokens without context, and 37,281 and 13,488 tokens with context, respectively. Although the common total above makes the retained logs operationally comparable, it is not a pure measure of model efficiency: the assistants used different tokenizers, agent harnesses, caching strategies, and tool-call patterns. These values are therefore reported as observed context traffic alongside elapsed time and the qualitative evaluation, rather than as an isolated performance ranking.

4.3 Cumulative Prototype Development (RQ2b)

Gemini was selected for the cumulative build as a pragmatic balance between the quality of the accepted result and the effort required to reach it. Its without-context run reproduced the intended visual direction, while its with-context run initially preserved an inappropriate scaffold theme but reached the intended result after design-focused follow-up prompts. Codex was faster but retained a visible spacing discrepancy, whereas Claude required substantially more elapsed time in the with-context condition. The selection therefore reflected suitability for the remaining project work; it does not establish that Gemini is generally superior to the other assistants.

After the bounded comparison, the prototype was developed as a separate cumulative build. Work was divided into small, dependent increments covering the shared navigation, operational overview, geographic drill-down, sensor monitoring, leak reporting and history, an assistant interface, and user settings. The task identifiers were useful for organizing the archive, but the analysis treats these increments primarily as parts of an evolving interface.

Development frequently continued across separate chat sessions. The stable `GEMINI.md` file was deliberately not used as a progress log; instead, the repository contained a separate `PROJECT_STATUS.md` file for evolving project state. At the beginning of later sessions, Gemini consulted this record to recover what had already been built. Each task prompt also required the assistant to update it before finishing by recording the completed increment, relevant shared components or mock-data changes, unresolved questions where applicable, and the next planned task. The researcher checked these updates as part of the per-task loop. This mechanism reduced the need to reconstruct the project history in every new conversation while keeping stable project guidance separate from temporary state. The status file remained a handoff aid rather than an authoritative source: its contents still had to be checked against the repository, and one stale task-folder reference was found during review.

The build order followed functional dependency. Shared navigation and layout were established before page-specific content. Overview and geographic views introduced common map, status, and drill-down behavior. Reporting and history then reused data conventions established earlier, while the assistant and profile features were integrated after the main monitoring workflow existed. This sequence made it possible to observe not only first-pass errors, but also whether earlier decisions remained valid when later features reused or revisited them.

For each increment, the researcher reviewed the task and design material, examined the assistant's proposed plan where the environment exposed one, and then reviewed the resulting implementation. Acceptance combined multiple forms of evidence: automated build and lint checks, interaction checks, comparison with the design, and inspection of the relevant source and assets. Corrections were issued

when the result departed from the intended scope or when later review exposed a problem in an earlier feature.

The later part of the build added a more explicit independent review stage. The building assistant's output was compared with the task, design, code, and available screenshots. This separation between building and reviewing was not uniform from the beginning, so it is treated as an observed process change, not as a controlled experimental factor.

4.4 Evidence and Analysis

The project archive contains task prompts, transcripts, plans, source code, screenshots, design references, evaluation records, and retrospective review reports. Primary artifacts were used to establish what was requested, built, checked, and corrected. Retrospective reports were used to locate candidate patterns, but claims were retained only when they could be checked against the underlying project record.

Analysis moved from individual episodes to recurring process patterns. An observation was treated as recurring when comparable evidence appeared in at least two separate parts of the build. Single episodes were retained only when they illustrated a distinct mechanism, such as a conflict between instruction layers or a failure to diagnose the underlying cause. The final synthesis grouped the evidence around four questions:

1. What changed when persistent context was present?
2. Which kinds of implementation failure recurred?
3. Which review practices prevented or reduced correction work?
4. Which lessons could be converted into reusable project guidance?

This thematic analysis produced the failure-pattern taxonomy, project-context blueprint, and supervised workflow presented in Chapter 5.

4.5 Methodological Boundaries

The bounded comparison used one run per condition, a fixed tool order, uncontrolled tool versions, and one researcher who was not blinded. It supports a comparative case description, not statistical generalization. The cumulative stage had no equivalent build without persistent context and therefore cannot establish that the context file caused its outcomes.

Documentation completeness also varied. A small number of workbook fields and task-log entries were unfinished or inconsistent, and some checks were not recorded uniformly. These gaps do not imply that the corresponding implementations were absent; they limit only the comparisons that depend on those fields. Chapter 5 therefore reports aggregate results only where the supporting record is sufficient, notes the remaining constraint once, and does not treat missing entries as negative outcomes.

Chapter 5

Results and Discussion

This chapter presents what the DigiWater project produced and what its development process revealed. The first outcome is the interface itself: an evolved Figma design and a working frontend prototype derived from the requirements and personas. The second outcome is a set of findings about AI-assisted development. These findings concern the practical value of persistent project context, recurring implementation failures, the role of human and AI review, and the kinds of guidance that should be carried from one task to the next.

Rather than reproducing the project log chronologically, the chapter organizes the evidence around these outcomes. Individual tasks are mentioned only when they provide a clear example of a broader finding.

5.1 Design and Prototype Outcome

The project produced two related artifacts. The first is a revised interface design that expresses the requirements and UX rationale established in Chapter 3. The second is a frontend-only React and TypeScript prototype implementing the principal navigation, overview, geographic drill-down, sensor, reporting, history, assistant, and profile surfaces.

Figures 5.1 and 5.2 show the entry point and system-wide overview. The geographic drill-down is shown in Figures 5.3 and 5.4; the reporting workflow appears in Figures 5.5 and 5.6; and the assistant and profile surfaces appear in Figures 5.7 and 5.8. Together, these figures present the complete interface sequence in workflow order. Each portrait page places one revised Figma screen in the first row and its corresponding implemented React screen in the second row, so that the related design and prototype evidence can be read together without rotating the page. The city and district examples also keep the Aqueduct and Marene navigation levels expanded, making the selected context and available district choices visible. The pairs are not intended as pixel-by-pixel comparisons because the design continued

to evolve as implementation and review exposed opportunities to improve the UX.

The design follows an overview-to-detail structure. Luca's operational needs are reflected in prominent status summaries, risk indicators, maps, and alerts, while Sara's analytical needs are supported through drill-down views, sensor history, filtering, and comparison-oriented content. Implementation also acted as a form of design inquiry. It revealed where annotations could be mistaken for interface copy, where a visually plausible component did not generalize across entities, and where later features exposed layout or consistency issues in earlier pages. The final design should therefore be read as the outcome of an iterative design–implementation–review process, not as a specification that remained unchanged from the beginning.

The project supports RQ1 at the level of requirements, design rationale, and a working prototype. It does not establish that the interface reduces cognitive load or improves performance for real operators, because no user study or task-based usability evaluation was conducted.

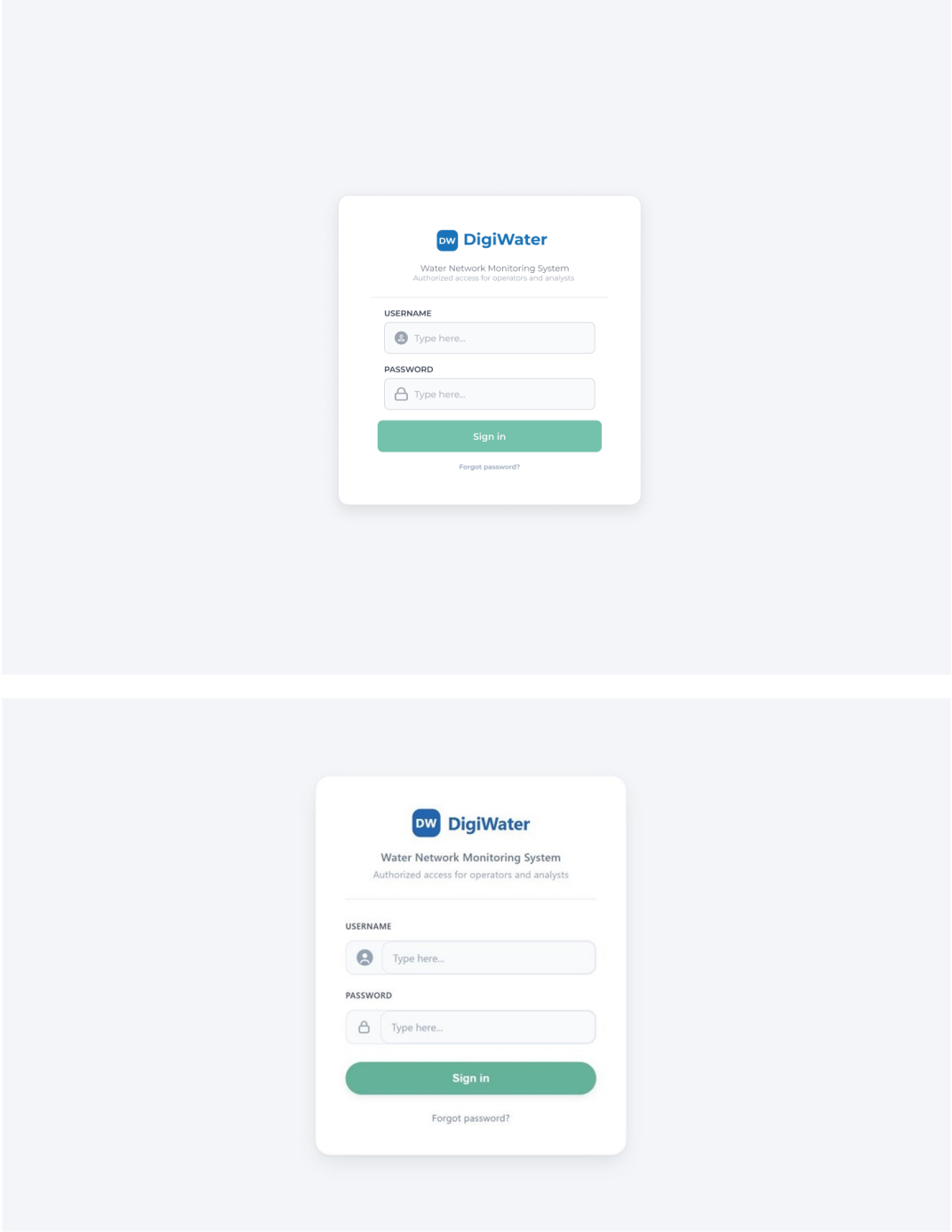


Figure 5.1: Login: Figma design (top) and React prototype (bottom).

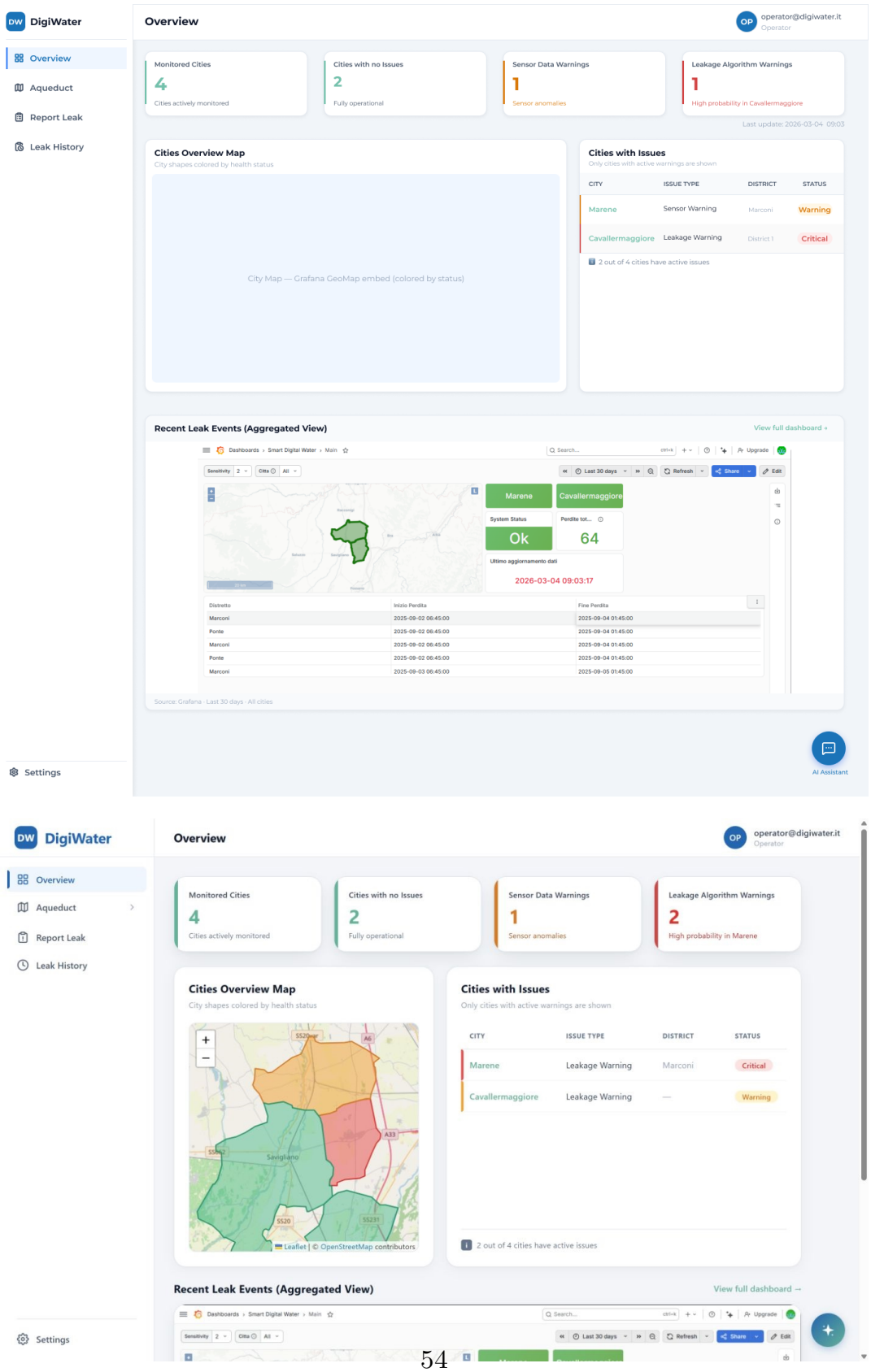


Figure 5.2: Overview: Figma design (top) and React prototype (bottom).

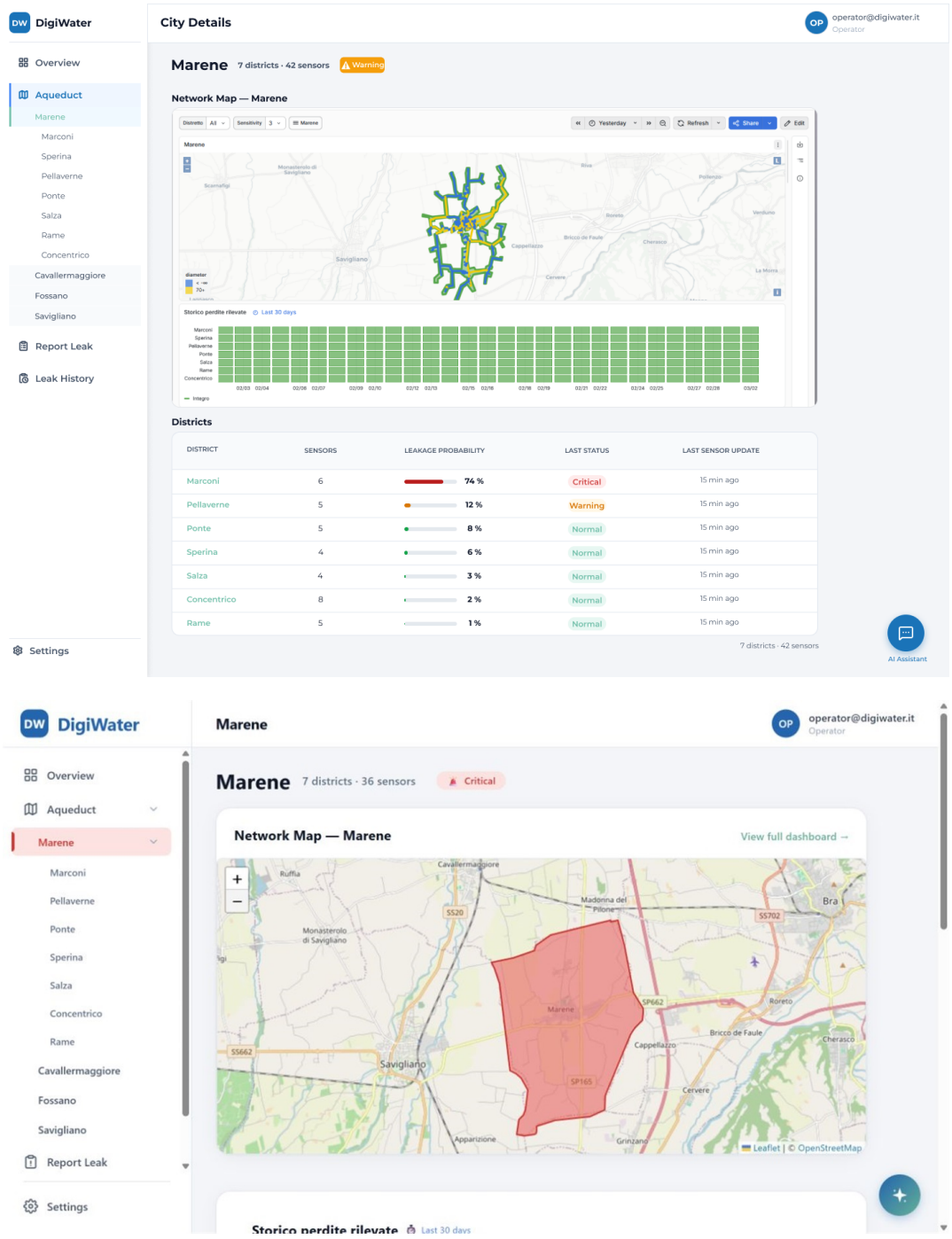


Figure 5.3: City details: Figma design (top) and React prototype (bottom).

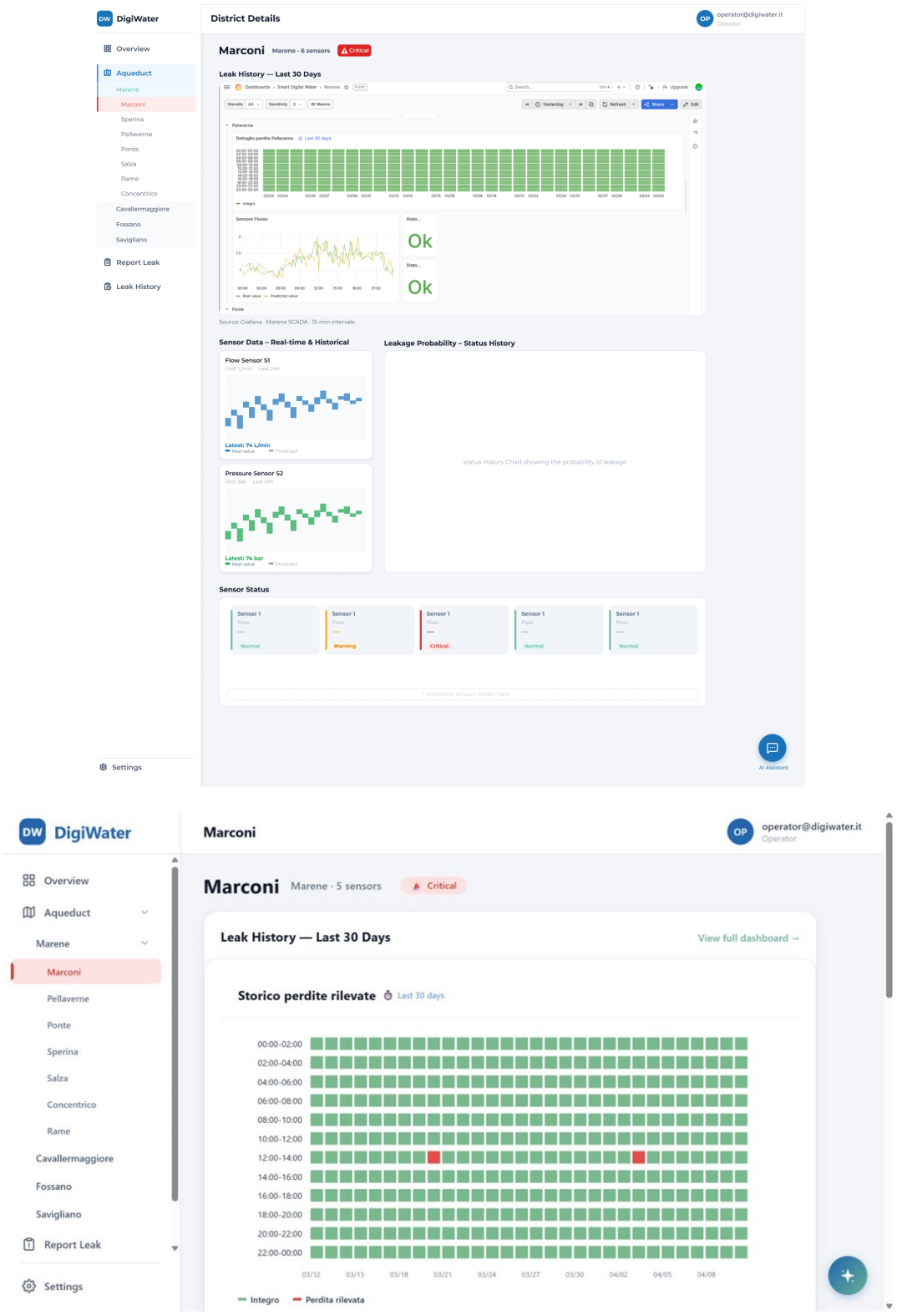


Figure 5.4: District details: Figma design (top) and React prototype (bottom).

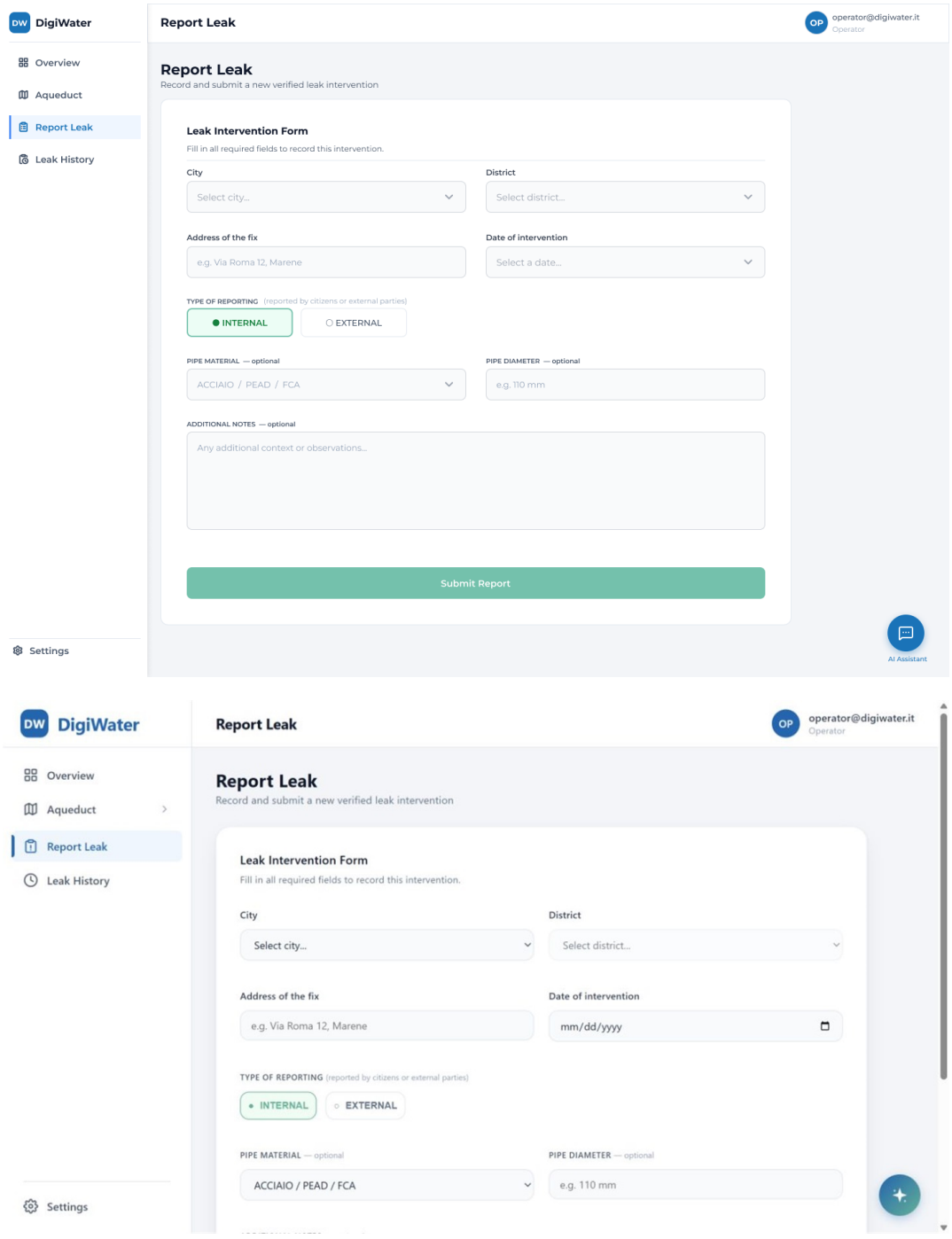


Figure 5.5: Leak reporting: Figma design (top) and React prototype (bottom).

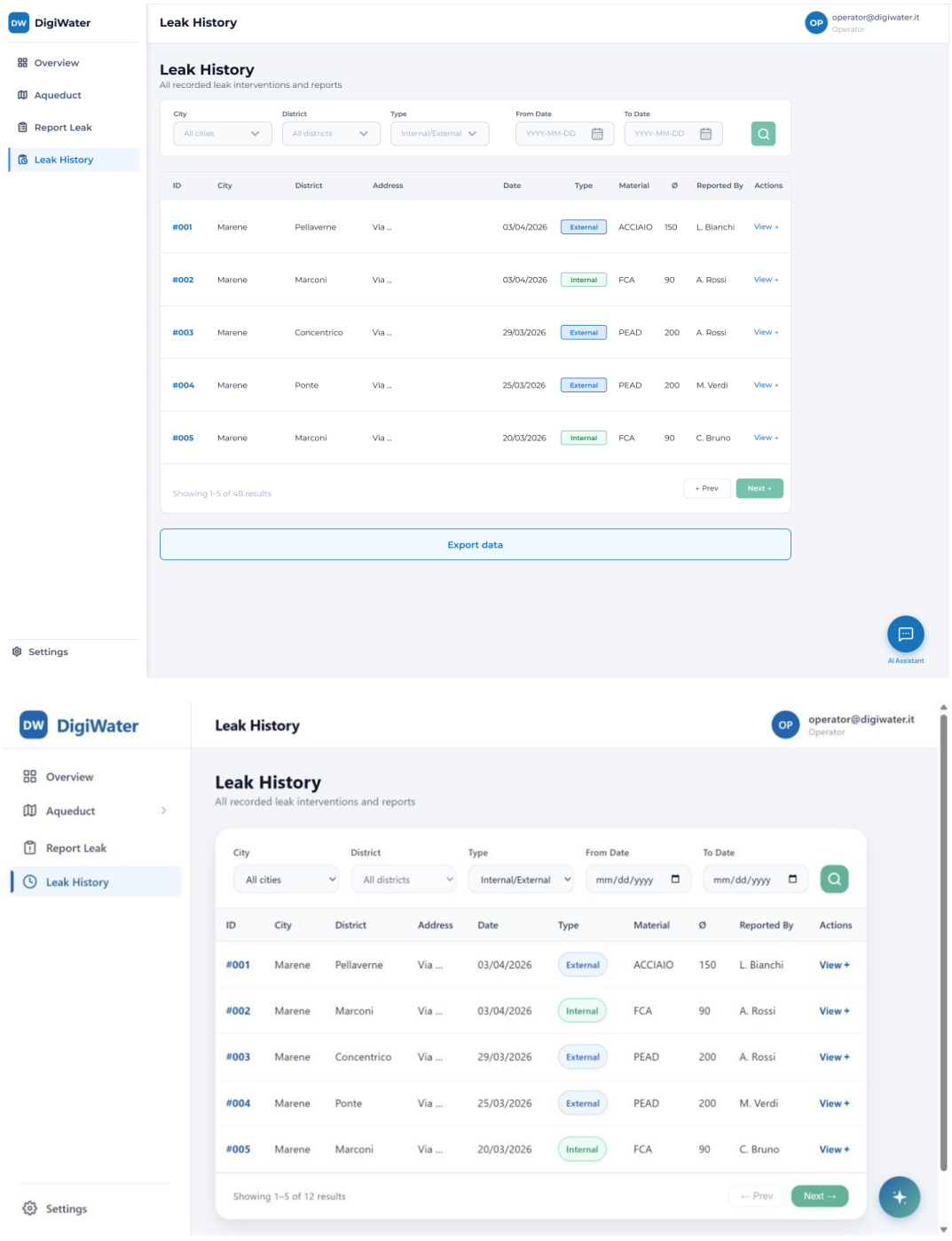


Figure 5.6: Leak history: Figma design (top) and React prototype (bottom).

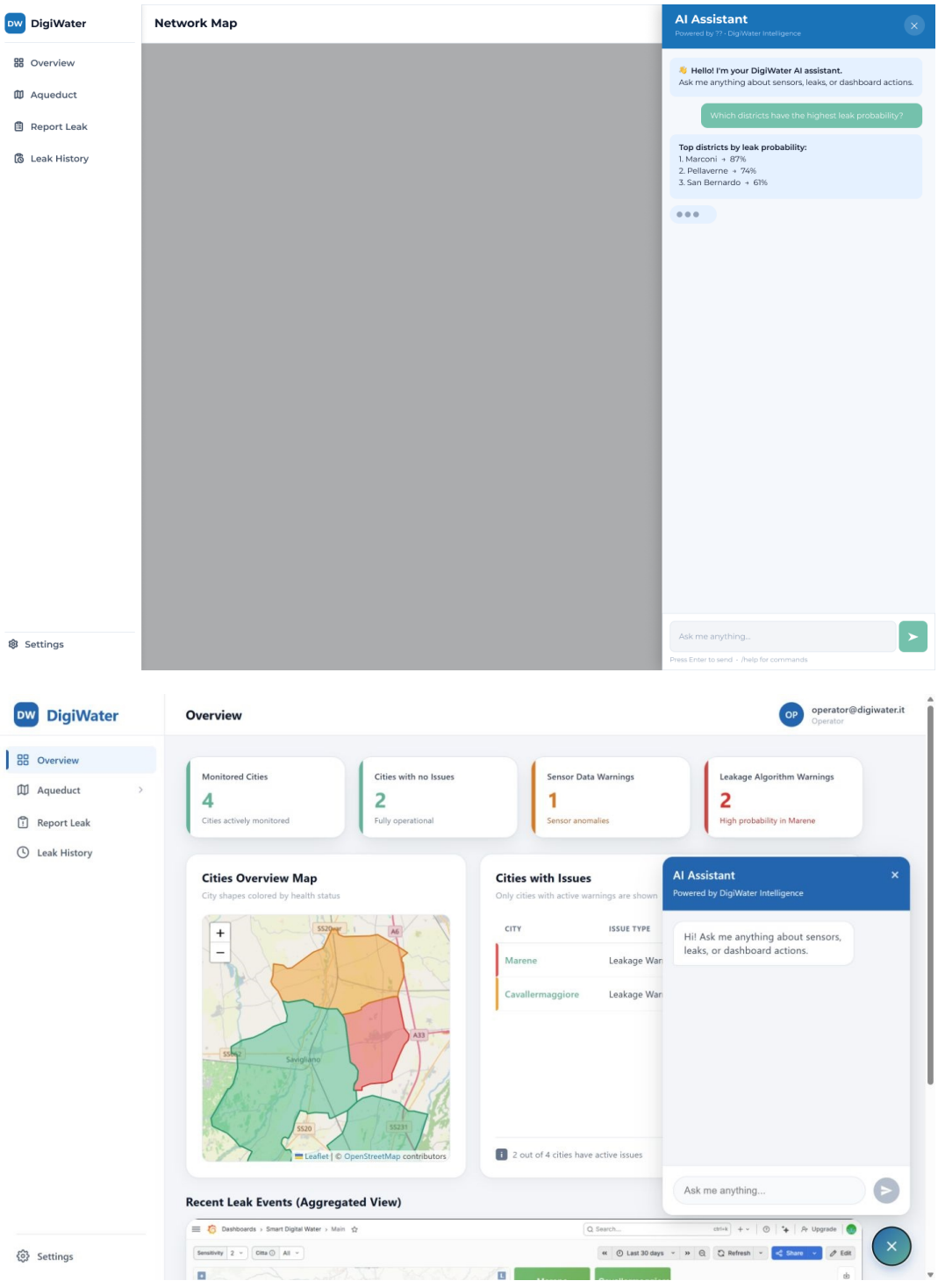


Figure 5.7: AI assistant: Figma design (top) and React prototype (bottom).

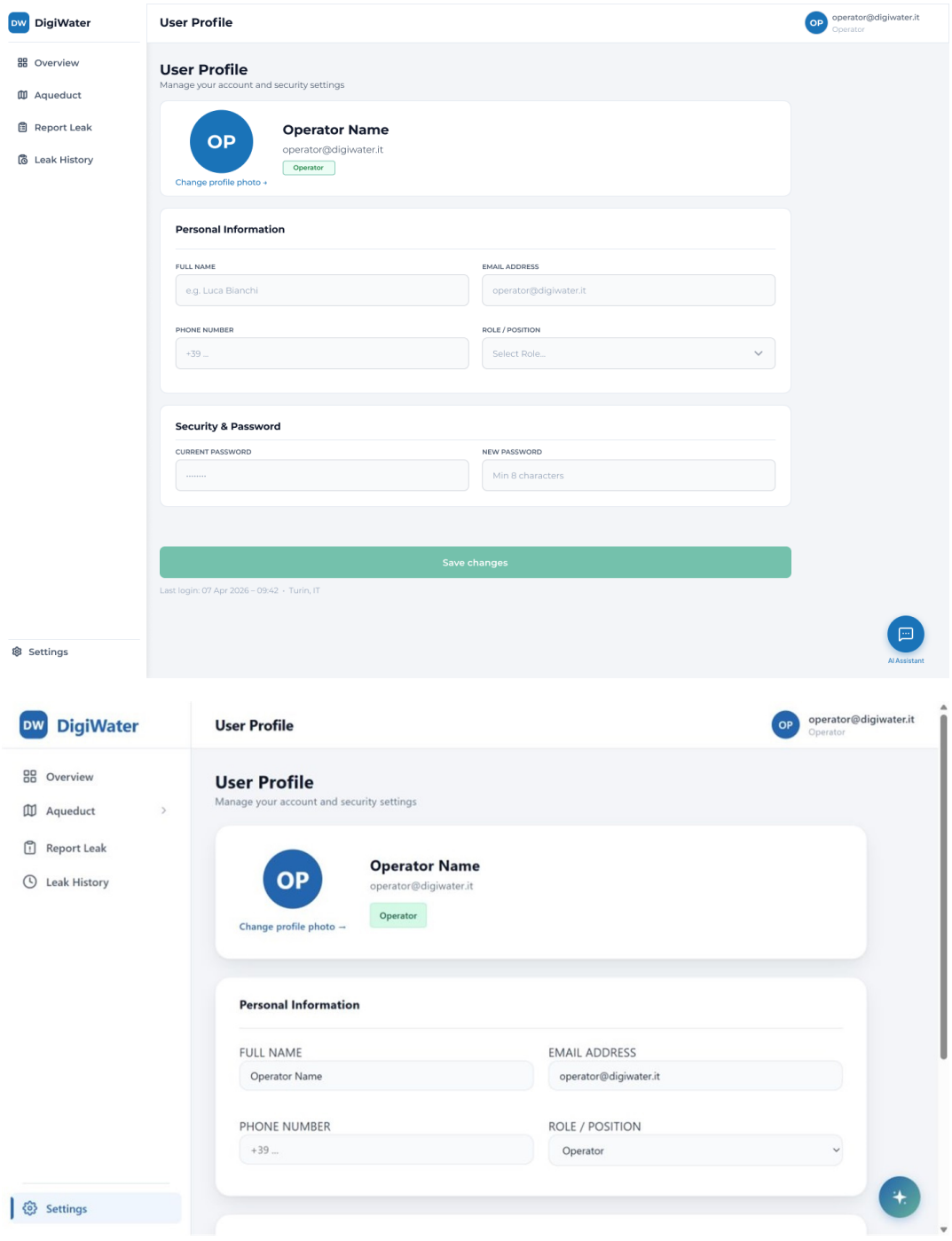


Figure 5.8: User profile: Figma design (top) and React prototype (bottom).

5.2 What Persistent Project Context Changed

The bounded comparison asked whether the same implementation task changed when the assistants received a persistent project-context file. Table 5.1 summarizes the recorded results.

Table 5.1: Task 02.1 comparison summary.

Assistant	Context	Req./Code/Visual	Time (min)	User prompts
Codex	Without	4/5/4	2.60	2
Codex	With	4/5/4	2.83	2
Claude	Without	5/5/3–4	8.83	2
Claude	With	5/5/3–4	24.50	3
Gemini	Without	5/5/5	8.22	1
Gemini	With	–/–/–	9.00	3

The Claude visual rating differs between two records, but the value is the same across its two conditions under either record. Gemini's with-context quality ratings were not completed because the record did not define clearly whether they should describe the initial output or the corrected final state. The initial result met the code and requirements expectations but required visual follow-up; the accepted result subsequently matched the intended direction. This prevents a full numerical comparison for that assistant but does not change the fact that all six conditions produced retained implementations.

For Codex and Claude, the recorded quality scores did not change between the two conditions. Codex also required the same correction in both conditions. This is a useful null result: adding persistent context did not automatically produce a visible improvement for a task that was already specified in detail. Because Gemini's with-context quality cells were unfinished, the same numerical claim cannot be made for all three assistants. No retrospective score was assigned because a single value would conceal the difference between the initial output and the accepted result after follow-up.

The more informative difference appeared in correction effort. The persistent file instructed assistants to extend the existing visual style, while the current task's design clearly required a different light and blue visual direction. Claude and Gemini initially preserved the scaffold theme when the context file was present. The rule was reasonable in isolation, but wrong for this task because it did not explain what should happen when a current design intentionally superseded an earlier convention. Codex, by contrast, showed the same spacing issue with and without context, on a topic the file did not cover.

The comparison therefore suggests a more precise conclusion than either "context files work" or "context files do not work." Persistent context is useful for stable

boundaries and conventions, but it is not a substitute for a specific task. Its benefit depends on whether its content is relevant, unambiguous, and compatible with the current instruction. When those layers conflict, additional context can increase rather than reduce correction work. This bounded result is consistent with broader evidence that repository-level instruction files do not automatically improve task success [6], while the present case adds a concrete mechanism: an otherwise sensible general rule can create a specific failure when its priority is unclear.

5.3 Observed Failure and Risk Patterns in the Cumulative Build

The cumulative build produced the complete prototype, but it also revealed several ways in which AI-generated code can be locally plausible and still wrong for the project. Table 5.2 synthesizes these observations into a case-derived taxonomy.

Table 5.2: Case-derived taxonomy of failure and risk patterns in AI-assisted implementation.

Pattern	Observed form	Practical implication
Instruction conflict	A stable project rule and the current design pointed in different directions.	Define instruction priority and require the conflict to be reported.
Under-generalization	A correction applied to one component did not transfer to a structurally similar page or sibling entity.	Promote recurring rules into persistent guidance or shared code.
Over-generalization	A narrowly requested visual change spread to an additional, unnamed element.	State both positive and negative scope: what changes and what must remain unchanged.
Generic-default substitution	Familiar defaults, such as a narrow form-page layout, reappeared despite the dashboard's full-width convention.	Record project-specific conventions explicitly and enforce them structurally where possible.
Example-to-rule confusion	An asset intended for one location was reused for other locations because it looked like a reasonable fallback.	Make entity-specific inputs explicit and prefer an honest empty state.
Design-annotation literalism	Explanatory text in a design reference was rendered as user-facing interface copy.	Distinguish design guidance from content requirements in the prompt.
Shallow diagnosis	Repeated property-level corrections were attempted before a basic structural error was found.	Run inexpensive syntax and structural checks before semantic debugging.
Isolated verification	A page passed its immediate review, but a defect was found only when related functionality was revisited later.	Add regression checks across shared layouts, components, and sibling pages.
Prompt-coverage gap	Some cross-feature issues had never been specified and were discovered only after multiple features coexisted.	Track emergent design issues separately from failures to follow a prompt.

5.3.1 Scope Transfer and Recurring Defaults

Two opposite behaviors appeared repeatedly. In some cases, a useful decision did not travel far enough: a corrected layout convention or entity-specific rule had to be rediscovered on another page. In other cases, a narrowly scoped change travelled too far because the assistant interpreted visual consistency as permission

to modify related elements. Together, these observations show that phrases such as “use the same treatment” or “make this green” leave the intended blast radius underspecified. Effective instructions name the target, the relevant precedent, and what must remain untouched.

The repeated return of a narrow page-width convention illustrates a second lesson. Conversational correction created local compliance, not durable project memory. Once the same issue has appeared more than once, another correction prompt is a weak response. The rule should be added to stable project guidance, represented by a shared component, or enforced through a check that makes recurrence difficult.

5.3.2 Verification and Delayed Discovery

Visual review and source inspection detected different classes of problem. Visual comparison was necessary for spacing, hierarchy, chart appearance, and scope. Source inspection was necessary for incorrect asset references, incomplete selectors, and hidden structural problems. Neither channel was sufficient alone.

In two documented episodes, a defect was found only when later work revisited related functionality. This is referred to here as detection latency: task acceptance occurred before the relevant relationship between pages or entities was exercised. A separate episode showed diagnostic latency: multiple reasonable styling corrections did not change the result because the actual cause was a structural syntax error elsewhere in the stylesheet. The practical lesson is simple. Verify across related features, and use cheap mechanical checks before investing in increasingly elaborate semantic explanations.

5.3.3 Correction Effort as a Process Signal

Among the nine cumulative increments with finalized correction ratings, one required no correction, four required minor correction, three required moderate correction, and one required major correction. The distribution is not a performance benchmark, but it shows that successful delivery often included substantive review after the first implementation. Three additional increments lacked a final rating and are excluded from this summary rather than treated as defect-free or unsuccessful.

The amount of correction was less informative than where it occurred. A correction made after coding required diagnosis, a new prompt or edit, another build, and renewed visual review. By contrast, several shared-state and scope issues were identified while reviewing an implementation plan, before code was written. This difference motivated a shift from treating review as final inspection to treating it as a sequence of earlier and later quality gates.

5.4 Practices Associated with Better Outcomes

The project record supports six practical observations about review and instruction quality.

1. **Clarifying the task before coding reduced later ambiguity.** A heavily discussed assistant-panel task resolved its shape, icon, content, and scope before implementation and did not require post-build correction on those dimensions. This is suggestive rather than controlled evidence, but it shows the value of prompt-stage review.

2. **Plan review caught cross-cutting problems cheaply.** In the later build, proposed behavior affecting navigation, sorting, and pagination was corrected before implementation. The same issues would have required a full build-and-fix cycle if found afterward.

3. **Explicit precedent was useful only when its relevant behavior was stated.** A later feature correctly reused an earlier data convention when the stored identifiers and required lookup were described precisely. Merely naming an earlier page did not guarantee that its conditional behavior would be preserved.

4. **Independent review changed the questions being asked.** The building assistant generally checked whether its chosen solution was internally coherent. The reviewer compared that solution with the prompt, design, related pages, and actual source references. This broader comparison exposed problems that a self-review of the immediate change did not.

5. **Not every later change was an implementation defect.** Some improvements addressed interactions that no original prompt had anticipated, while one correction reflected a reviewer changing the desired scope. Separating specification deviations from emergent design improvements produces a fairer and more useful account of AI performance.

6. **Stable guidance and evolving project state served different purposes.** The fixed `GEMINI.md` file carried durable constraints, while the separate status file
`PROJECT_STATUS.md` carried completed work, shared-data changes, unresolved questions, and the next task across separate chat sessions. This reduced repeated explanation, but the status record still required verification against the repository because a stale path was found during review.

These observations also clarify requirements traceability. Personas were most useful when converted into a checkable design choice, and task-to-task knowledge

was most reliable when the relevant implementation detail was verified and restated. Traceability did not occur automatically because a context file existed; it was maintained through explicit prompts, review, and acceptance decisions.

5.5 A Case-Derived Blueprint for Project-Context Files

The study began with an existing project-context file. Its more important contribution is the set of design principles that can be derived from what the file covered, omitted, and occasionally contradicted. Table 5.3 presents a blueprint for future AI-assisted frontend projects.

Table 5.3: Case-derived blueprint for a persistent project-context file.

Section	**Recommended content**
Purpose and boundaries	The product goal, intended users, prototype or production status, and explicit out-of-scope work.
Instruction hierarchy	Which source wins when the current task, design, requirements, existing implementation, and persistent conventions disagree.
Architecture and tools	Stable stack decisions, repository organization, approved dependencies, and genuine non-negotiable constraints.
Domain and UX principles	Only durable principles that affect many tasks, linked to the needs they serve.
Project-specific conventions	Rules learned from recurring failures, stated concretely enough to verify.
Scope behavior	A requirement to change only named targets, preserve unrelated behavior, and report any necessary expansion of scope.
Verification contract	The minimum structural, automated, behavioral, visual, and regression checks required before claiming completion.
Conflict and uncertainty protocol	A requirement to surface contradictions, missing evidence, and unverified claims rather than silently guessing.

The blueprint is most useful as part of a three-layer project-memory model. First, the persistent context file stores stable knowledge that should remain valid across many tasks. Second, a separate, updateable project-status file records the evolving implementation state and supports handoff between chat sessions. Third, the current task prompt and design reference define the immediate goal and acceptance conditions. Keeping these layers separate avoids turning the context file into an ever-growing diary while still allowing a new assistant session to recover the project's current state quickly.

For DigiWater, the evidence supports several concrete examples of durable guidance:

- When the current task or supplied design explicitly changes an earlier visual convention, follow the current task and report the conflict.

- Dashboard page containers use the available content width; do not add a fixed maximum width unless the task explicitly requires one.

- Do not reuse an entity-specific image or dataset for sibling entities; show an unavailable state when corresponding evidence does not exist.

- Treat design annotations as developer guidance unless they are explicitly identified as interface copy.

- Apply a correction only to the named targets and leave unrelated elements unchanged.

- If a correction does not affect the observed result, reassess the root cause and run structural checks before repeating the same type of fix.

The blueprint is deliberately not a progress log. Current task details, temporary values, completion status, and one-off design measurements belong in task prompts or the separate project-status record. Keeping those concerns separate makes the persistent file shorter, more stable, and less likely to compete with the current specification.

The strongest rules should not necessarily remain prose. When a convention can be represented by a shared component, required type, lint rule, or automated test, structural enforcement is more reliable than asking an assistant to remember a correction. The context file should explain the rule; the codebase should prevent its recurrence where practical.

5.6 A Supervised Workflow for AI-Assisted Development

The combined findings support the three-stage workflow shown in Figure 5.9. It is a synthesis of the case rather than a process that was followed uniformly from the beginning.

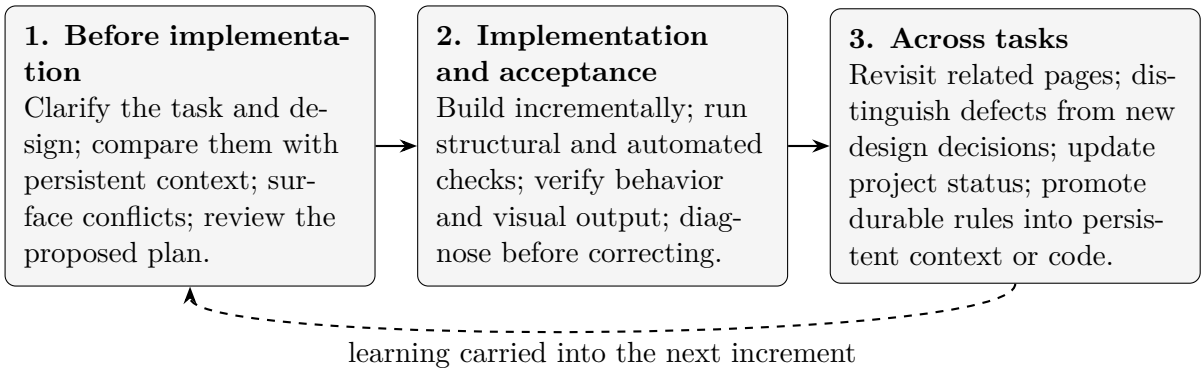


Figure 5.9: Case-derived supervised workflow for AI-assisted development.

The workflow makes the human contribution visible. The researcher or developer does not merely issue prompts; they interpret requirements, decide scope, resolve conflicts, assess evidence, and determine when a result is acceptable. AI can support both building and reviewing, but review is stronger when it is independent of the assumptions that produced the first implementation.

At the end of each increment, the status record should capture what changed and what comes next; only lessons expected to remain valid should be promoted into the persistent context file, and recurring rules should be enforced in code where practical.

5.7 Implications and Limitations

For RQ2a, the recorded comparison shows no quality-score difference for the two assistants with complete paired ratings. It also shows that a persistent instruction can create additional effort when its relationship with the current task is unclear. These findings are bounded to one task and one run per condition; they do not establish equivalence between conditions or a general effect across tools and projects.

For RQ2b, the cumulative build shows that the practical challenge was not simply generating code. It was maintaining scope, precedent, visual intent, and verification across an evolving codebase. Persistent context helped state stable boundaries, but task-level specificity, early review, independent verification, and cross-task learning remained necessary. The cumulative stage had no no-context comparison, so it cannot show that the context file caused the observed outcomes.

The evidence has additional limits. Quality and correction ratings were made by one researcher, review practice changed during the project, and not every log field or check was completed uniformly. Incomplete entries were excluded from the particular comparisons that required them; they were not interpreted as failed implementations. The final taxonomy, blueprint, and workflow are therefore case-derived contributions to be evaluated in future work, not validated universal prescriptions.

Within those boundaries, the thesis produces a result that is more useful than a chronological record of corrections. It identifies why seemingly helpful instructions can fail, how recurring AI behaviors can be recognized, and how an AI-assisted project can turn those observations into durable guidance and a more reliable development process.

Chapter 6

Conclusions

6.1 What the Thesis Achieved

This thesis used the development of DigiWater to connect requirements engineering, UX design, frontend implementation, and the study of AI-assisted software development. It produced three overarching contributions, realized through five concrete outputs.

First, the stakeholder material and the Luca and Sara personas were translated into a requirements and UX specification for a water-network monitoring interface. The design balances rapid operational awareness with access to deeper analytical detail through a consistent overview-to-detail structure.

Second, the work produced an evolved Figma design and a working frontend prototype. The two artifacts played different roles. Figma expressed and revised the intended UX, while the React and TypeScript implementation tested how that design behaved as an integrated interface. Implementation findings, visual review, and supervisor feedback contributed to later design decisions.

Third, the implementation process provided a bounded comparison of three AI coding assistants with and without persistent project context, followed by a longitudinal record of cumulative development. This evidence showed that persistent instructions are not automatically beneficial: their value depends on their relevance, specificity, and relationship with the current task.

This case-study contribution is expressed through three methodological outputs: a taxonomy of observed implementation failures and risks, a blueprint for persistent project-context files, and a supervised workflow for AI-assisted development.

6.2 Answers to the Research Questions

RQ1. The DigiWater interface addresses rapid operational awareness by prioritizing status summaries, maps, alerts, and clear severity indicators on overview

surfaces. It supports analytical exploration through geographic drill-down, sensor history, filtering, and detailed status views. The interface was refined through an iterative relationship between design, implementation, and review. This answers RQ1 at the level of requirements and design rationale. User-facing benefits such as reduced cognitive load or faster decision-making remain to be evaluated with representative operators and engineers.

RQ2a. For the two assistants with complete paired quality ratings, adding persistent project context produced no observed score difference on the bounded task. The more revealing result concerned correction effort: when a general style-preservation instruction conflicted with the current design, assistants could follow the persistent rule and move away from the intended result. The comparison therefore suggests that persistent context is not valuable simply because it exists. It must contain stable, relevant rules and define how conflicts with more specific instructions should be handled.

RQ2b. Across the cumulative build, the main challenge was maintaining design intent, scope, and implementation knowledge as the interface grew. The record revealed recurring under- and over-generalization, reuse of generic defaults, example-to-rule confusion, literal interpretation of design annotations, shallow diagnosis, and defects discovered only when related features were revisited. It also showed that prompt clarification, plan review, explicit precedent, independent review, structural checks, and cross-task regression testing could reduce or expose these problems. Persistent context supported stable boundaries, but it did not replace these practices.

6.3 Original Contributions

The three overarching contributions introduced in Chapter 1 are expressed here as five concrete outputs:

1. **A requirements-driven DigiWater interface specification.** The design connects operational and analytical user needs to a coherent monitoring workflow.

2. **An evolved design and implemented frontend prototype.** Together, these artifacts demonstrate how the proposed requirements can be expressed visually and implemented as an interactive system.

3. **A case-derived taxonomy of AI implementation failures.** The taxonomy turns individual corrections into recognizable mechanisms that can be monitored in other incremental AI-assisted projects.

4. **A blueprint for persistent project-context files.** The blueprint defines what stable context should contain, what should remain task-specific, how instruction priority should be stated, and how recurring failures can be converted into durable project rules.

5. **A supervised AI-assisted development workflow.** The workflow integrates prompt and plan review, implementation and acceptance checks, independent diagnosis, and cross-task learning.

The last three outputs together constitute the thesis's third overarching contribution and are intentionally described as case-derived. They are supported by the DigiWater record but have not been validated as universal or optimal. Their value is that they make the project's lessons explicit, testable, and reusable rather than leaving them dispersed across conversation logs and correction reports.

6.4 Future Work

Future work should evaluate the prototype with representative DigiWater users. Operational status-recognition and analytical drill-down tasks could measure usability, task performance, and cognitive load, while qualitative feedback could test whether the interface communicates risk and uncertainty clearly.

The AI-assisted development findings should also be tested on a new cumulative build. A stronger study would repeat the same multi-task sequence with and without persistent project context, use multiple runs per condition, control tool versions, rotate run order, and apply blinded evaluation where practical.

The proposed context-file blueprint can be evaluated independently. Future experiments could compare a minimal context file, a long descriptive file, and a conflict-aware file following the blueprint in Chapter 5. This would test whether instruction hierarchy, explicit scope behavior, and a verification contract reduce correction effort without unnecessarily constraining the assistant.

Finally, some recurring conventions should be moved from prose into the codebase. Shared layout components, stronger types, lint rules, and regression tests can make a project less dependent on either a person or an assistant remembering a previous correction.

6.5 Closing Remarks

DigiWater demonstrates both the promise and the limits of AI-assisted frontend development. The assistants accelerated the production of a substantial prototype, but reliable progress depended on more than code generation. It required clear task

boundaries, careful interpretation of design evidence, human judgment, independent verification, and the deliberate conversion of recurring mistakes into project knowledge.

The central conclusion is therefore not that persistent context succeeds or fails in isolation. Persistent context is one layer in a supervised development process. It is most useful when it is concise, conflict-aware, and limited to durable knowledge; when current tasks remain explicit; and when claims of completion are verified across code, behavior, and design. By making those conditions visible, the thesis provides both DigiWater with a concrete interface direction and future AI-assisted projects with a clearer method for working responsibly and effectively.

Appendix A

Evaluation Instrument and Evidence Archive

This appendix summarizes the evaluation material used in the AI-assisted development study. The complete workbook, frozen prompts, transcripts, repositories, screenshots, and review reports are retained in the accompanying digital research archive. They are not reproduced in full here because their purpose is auditability rather than development of the thesis argument.

A.1 Evaluation Structure

The comparative study used a common record for all six conditions. Table A.1 groups the recorded fields by analytical purpose.

Table A.1: Summary of the comparative evaluation instrument.

Category	**Recorded information**
Identification	Assistant, instruction condition, run order, date, and repository reference.
Effort	Elapsed time, correction rating, manual intervention, clarification, and follow-up prompts.
Technical verification	Development server, build, lint, runtime behavior, and console observations.
Implementation outcome	Files changed, completion, acceptance criteria, and final decision.
Quality assessment	Requirements compliance, code health, and visual and UX fidelity on five-point ordinal scales.
Context adherence	Researcher assessment of adherence to the persistent project-context file in the with-context conditions.
Supporting evidence	Screenshot reference and explanatory notes.

Prompt totals reported in Chapter 5 were checked against the conversation transcripts rather than derived from the workbook's inconsistently encoded prompt-iteration field. Table A.2 records the resulting counts.

Table A.2: Transcript-derived prompts in the six-condition comparison.

Assistant	**Context**	**Initial**	**Build/runtime**	**Design**	**Total**
Codex	Without	1	0	1	2
Codex	With	1	0	1	2
Claude	Without	1	0	1	2
Claude	With	1	0	2	3
Gemini	Without	1	0	0	1
Gemini	With	1	0	2	3

A.2 Data-Completeness Note

All six comparison conditions produced retained implementations. A limited number of evaluation fields were nevertheless incomplete or inconsistent. Gemini's with-context quality ratings were not finalized because the record did not specify whether the three scores should represent the initial output or the corrected final state. The initial result met the code and requirements expectations but required visual follow-up before reaching the accepted result. Console verification was not documented consistently for the Gemini comparison runs, and two records disagree on Claude's visual score, although both records give the same value to its two conditions. In the cumulative log, three increments lacked a final correction-effort rating.

These gaps affect only the comparisons that depend on the missing fields. Accordingly, the thesis does not infer missing values, does not count an unfinished rating as a failure or a zero, and states numerical findings using the available denominator. The complete unreconciled rows remain available in the original workbook for audit.

A.3 Digital Archive Organization

The accompanying archive separates evidence by purpose:

- **Methodology material:** persistent context files, task specifications, and evaluation workbook;
- **Comparative runs:** six repositories, transcripts, screenshots, and assistant-specific reports;

- **Cumulative build:** the evolving application repository, task prompts, implementation reports, and correction records;
- **Review material:** cross-task review reports used to identify candidate patterns and verify them against primary artifacts;
- **Design material:** initial and revised Figma exports, design references, and supporting Grafana images.

This separation preserves traceability without requiring the main thesis to repeat operational logs or source-file histories.

Appendix B

Task and Evidence Index

The cumulative build was managed through numbered prompts so that each change could be traced to its specification and evidence. Table B.1 provides a compact index. The complete frozen wording is retained in the digital archive.

Table B.1: Functional index of cumulative-build task material.

Area	**Archived task IDs**	**Main evidence**
Login and routing	01.1	Prompt, implementation report, source, and final design reference.
Navigation and overview	02.1–02.4	Comparative and cumulative prompts, process report, map and status implementation, correction record, and design references.
City monitoring	03.1–03.2	Prompt, session and review reports, map and district-table source, and visual evidence.
District monitoring	04.1–04.3	Prompt, session and review reports, sensor and history implementation, correction evidence, and design references.
Leak reporting	05.1	Prompt, form implementation, plan review, correction record, and design reference.
Leak history	06.1	Prompt, table and filtering implementation, plan review, correction record, and design reference.

Table B.1 – continued

Area	**Archived task IDs**	**Main evidence**
AI assistant	07.1	Iterated prompt, implementation report, review evidence, and open/closed design references.
Profile and settings	08.1	Prompt, implementation report, structural and visual correction evidence, and design reference.

The identifiers in this table are archive keys, not the conceptual results of the thesis. Chapter 5 therefore refers to them only where an individual episode is necessary to support a broader finding.

Bibliography

[1] European Commission, Directorate-General for Environment. *Poorly Managed Water*. No publication date given on source page; states that global freshwater demand is predicted to exceed supply by 40% by 2030 and that an average of 23% of treated water in the EU is lost during distribution. URL: https://environment.ec.europa.eu/topics/water/water-wise-eu/poorly-managed-water_en (visited on Aug. 26, 2026).

[2] Hatem Taha AbuEltayef, Khaldoun S. AbuAlhin, and Khalil M. Alastal. «Addressing Non-Revenue Water as a Global Problem and Its Interlinkages with Sustainable Development Goals». In: *Water Practice and Technology* 18.12 (2023), pp. 3175–3202. DOI: 10.2166/wpt.2023.157.

[3] Politecnico di Torino. *Optimising Water Resources Management with Artificial Intelligence*. July 31, 2025. URL: https://www.polito.it/en/polito/communication-and-press-office/poliflash/optimising-water-resources-management-with-artificial (visited on Sept. 3, 2026).

[4] agents.md Community. *AGENTS.md: A Simple, Open Format for Guiding Coding Agents*. URL: https://github.com/agentsmd/agents.md (visited on Aug. 11, 2026).

[5] GitHub. *How to Write a Great AGENTS.md: Lessons from over 2,500 Repositories*. 2025. URL: https://github.blog/ai-and-ml/github-copilot/how-to-write-a-great-agents-md-lessons-from-over-2500-repositories/ (visited on Aug. 11, 2026).

[6] Thibaud Gloaguen et al. «Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents?» In: *arXiv preprint arXiv:2602.11988* (2026). arXiv: 2602.11988. URL: https://arxiv.org/abs/2602.11988.

[7] Mica R. Endsley. «Toward a Theory of Situation Awareness in Dynamic Systems». In: *Human Factors* 37.1 (1995), pp. 32–64. DOI: 10.1518/001872095779049543.

[8] John Sweller. «Cognitive Load During Problem Solving: Effects on Learning». In: *Cognitive Science* 12.2 (1988), pp. 257–285. DOI: 10.1207/s15516709cog1202_4.

[9] John Sweller, Jeroen J. G. van Merriënboer, and Fred G. W. C. Paas. «Cognitive Architecture and Instructional Design». In: *Educational Psychology Review* 10.3 (1998), pp. 251–296. DOI: 10.1023/A:1022193728205.

[10] Rong Fang et al. «A Systematic Review of Empirical Studies on Situation Awareness: Perspectives From the Interaction Among Humans, Machines, and the Task Environment». In: *PsyCh Journal* 14.5 (2025), pp. 718–733. DOI: 10.1002/pchj.70027.

[11] Nuo Cheng et al. «The Influence of Learning Analytics Dashboard Information Design on Cognitive Load and Performance». In: *Education and Information Technologies* 29.15 (2024), pp. 19729–19752. DOI: 10.1007/s10639-024-12606-1.

[12] Corentin Burnay, Mathieu Lega, and Sarah Bouraga. «Business Intelligence and Cognitive Loads: Proposition of a Dashboard Adoption Model». In: *Data & Knowledge Engineering* 152 (2024). DOI: 10.1016/j.datak.2024.102310.

[13] EEMUA. *Alarm Systems – A Guide to Design, Management and Procurement*. EEMUA Publication 191. Engineering Equipment and Materials Users Association, 2024. URL: https://www.eemua.org/products/publications/print/eemua-publication-191 (visited on Aug. 11, 2026).

[14] EEMUA. *Contents – EEMUA Publication 191, 4th Edition (November 2024)*. Publicly available table of contents, cited only to confirm the scope of topics covered, not their detailed content. 2024. URL: https://www.eemua.org/getattachment/9d3f8071-55c3-49bf-a74a-3bf6ad4a2e0f/Contents-EEMUA-Publication-191-Edition4-November-2024.pdf (visited on Sept. 1, 2026).

[15] Andy Brazier. «Alarm Management – Update to EEMUA 191». In: *Hazards 35 Symposium Series No. 172*. Prepared on behalf of EEMUA; freely accessible conference paper summarizing the EEMUA Publication 191 fourth-edition update. Institution of Chemical Engineers (IChemE), 2025. URL: https://abrisk.co.uk/wp-content/uploads/2025/12/2025-Hazards-35-Alarm-Management-EEMUA-191-update.pdf (visited on Sept. 1, 2026).

[16] Ben Shneiderman. «The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations». In: *Proceedings of the 1996 IEEE Symposium on Visual Languages*. Washington, DC, 1996, pp. 336–343.

[17] John Pruitt and Jonathan Grudin. «Personas: Practice and Theory». In: *Proceedings of the 2003 Conference on Designing for User Experiences (DUX '03)*. ACM, 2003, pp. 1–15. DOI: 10.1145/997078.997089.

[18] International Organization for Standardization. *ISO 9241-210:2019 – Ergonomics of Human-System Interaction – Part 210: Human-Centred Design for Interactive Systems*. Tech. rep. International Organization for Standardization, 2019. URL: https://www.iso.org/standard/77520.html (visited on Aug. 12, 2026).

[19] Orlena C. Z. Gotel and Anthony Finkelstein. «An Analysis of the Requirements Traceability Problem». In: *Proceedings of the 1st IEEE International Conference on Requirements Engineering (ICRE '94)*. Colorado Springs, CO, 1994, pp. 94–101. DOI: 10.1109/ICRE.1994.292398.

[20] Robin Wieruch. *React Folder Structure Best Practices*. 2026. URL: https://www.robinwieruch.de/react-folder-structure/ (visited on Aug. 11, 2026).

[21] Kent C. Dodds. *Colocation*. 2019. URL: https://kentcdodds.com/blog/colocation (visited on Aug. 11, 2026).

[22] alan2207 and contributors. *Bulletproof React: A Simple, Scalable, and Powerful Architecture for Building Production-Ready React Applications*. URL: https://github.com/alan2207/bulletproof-react (visited on Aug. 11, 2026).

[23] Miku Watanabe et al. «On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub». In: *ACM Transactions on Software Engineering and Methodology* (2026). DOI: 10.1145/3798166. arXiv: 2509.14745.

[24] Hartwig Grabowski. «The Spec Growth Engine: Spec-Anchored, Code-Coupled, Drift-Enforced Architecture for AI-Assisted Software Development». In: *arXiv preprint arXiv:2606.27045* (2026). Solo-authored preprint, not peer-reviewed; no empirical evaluation reported. arXiv: 2606.27045. URL: https://arxiv.org/abs/2606.27045.