



**Università
di Genova**

University of Genoa

**Department of Electrical, Electronic, Telecommunications Engineering, and
Naval Architecture (DITEN)**

A Human-Factor-Centered Framework for Automated Evaluation of Industrial Digital Platforms

Author:

Parmida Shahani (6270122)

Supervisors:

Prof. Francesco Bellotti

Prof. Riccardo Berta

Co-Supervisors:

Dr. Hadise Rojhan

Academic Year: 202 – 2026

**Thesis submitted in partial fulfillment of the requirements for the degree of
*Master of Engineering Technology for Strategy (and Security)***

Acknowledgments

This thesis was developed as part of the SYNERGIES project, supported by the Horizon Europe framework under Grant Agreement No. 101146542. I sincerely thank my supervisors, Prof. Francesco Bellotti and Prof. Riccardo Berta. Their guidance and clear feedback were incredibly helpful, and I am very grateful for the time they spent supporting my work.

A special thank you goes to Dr. Hadise Rojhan. She guided me through building the AUREA framework from the very beginning. Without her help and hard work, we could not have published and presented our paper at the SEEDA CECNSM conference in Greece. I also thank the rest of the research team for their teamwork and good ideas.

I would like to thank Prof. Agostino Bruzzone, Head of the Master Degree Programme in Engineering Technology for Strategy and Security, for his support. Thank you to the University of Genoa for giving me a great place to study and work. I am also grateful to the open source software community for the tools that made this project possible.

Finally, a big thank you to my family and friends. To everyone who supported me here in Italy and back home, your patience and encouragement mean the world to me.

Abstract

Digital platforms must be usable, secure, and compliant with regulations such as the GDPR, but their evaluation is usually fragmented across various tools and methods. This thesis presents a comprehensive investigation into unified platform evaluation, starting from my participation in analyzing the first release of the SYNERGIES platform. While SYNERGIES was technically robust (112 ms average response time with 3000 concurrent users), its usability was weak (SUS 59.9, NPS negative 27.7, and only 14 percent of participants completing tasks without help). Furthermore, the evaluation required multiple disconnected tools and heavily depended on manual expert checklists.

To address these limitations, I designed AUREA (Automated Unified Runtime Auditor), a framework that audits a public web platform from the outside in a single browser session. The methodology combines Playwright for runtime data collection, Python rules for direct measurements, and a local Llama 3 model run through Ollama for context dependent checks. In a pilot study, AUREA was executed 25 times on the public pages of PagoPA and Just Eat UK on a low-end laptop, using strictly harmless requests of the 16 dynamically evaluated runtime indicators, 10 passed for PagoPA and 9 for Just Eat (the overarching dashboard score out of 21 intrinsically includes five fixed baseline constants).

An audit took 602 s and 438 s respectively, with over 93 percent of the execution time spent in the generative AI step.

Key contributions include: (1) the design of a hybrid auditing pipeline that consolidates fragmented evaluation methods into a single external session, (2) the integration of a local Large Language Model to evaluate context dependent factors such as privacy transparency, (3) the successful execution of the automated script across two diverse real world platforms without manual code changes, and (4) a detailed performance profiling that highlights the computational time cost of local generative AI.

The results demonstrate that combining web automation with local AI models provides a viable path for unified platform auditing, although it is currently constrained by execution time and the proxy nature of the checks. The thesis concludes by listing eleven specific limits of the current implementation and provides strategic recommendations for future work, prioritizing validation against domain experts and the extension of the framework to authenticated areas.

Keywords: digital platforms, automated auditing, human factors, large language models, Llama 3, web automation, privacy compliance, usability evaluation, SYNERGIES

Table of Contents

Contents

Acknowledgments	2
Abstract	3
Table of Contents	4
List of Figures	6
List of Tables	7
List of Abbreviations and Symbols	8
Chapter 1 - Introduction	10
1.1 Background and Motivation	10
1.2 Problem Statement	11
1.3 Motivation	12
1.4 Proposed Solution and Research Questions	14
1.5 Scope of the Thesis	15
1.6 Thesis Structure	16
Chapter 2 - Literature Review and Related Work	17
2.1 Overview of Digital Platform Evaluation	17
2.2 Usability and Privacy Auditing Principles	17
2.3 Large Language Models in Software Testing	18
2.4 SYNERGIES Project Context	18
2.5 Integrated Frameworks and the Gap	19
2.6 The SYNERGIES Context	21
Chapter 3 - Problem Identification and Case Study	23
3.1 The SYNERGIES Platform and Its Evaluation	23
3.2 Usability Results	24
3.3 Technical Performance and Interoperability	27
3.4 Expert-Based Assessment	29
3.5 Limits of the Evaluation Process	29
4.1 Design Principles and Requirements	31
4.2 Architecture	32

4.3 Output and Data Model.....	33
4.4 What AUREA Covers and What It Does Not.....	34
4.5 Data Acquisition and Human in the Loop Integration.....	35
4.6 Semantic Reasoning and Deterministic Fallbacks	36
Chapter 5 - Implementation and Experimental Evaluation	38
5.1 Implementation Overview	38
5.2 Runtime Data Acquisition.....	39
5.3 Deterministic Checks	40
5.4 Semantic Checks.....	42
5.5 Reporting to Measurify.....	43
5.6 Experimental Setup	44
5.7 Results.....	45
5.8 Validation of Semantic Output	49
5.9 Limits of the Current Implementation	50
Chapter 6 - Discussion and Threats to Validity	52
6.1 Answers to Research Questions.....	52
6.2 Discussion	54
6.3 Threats to Validity	55
Chapter 7 – Conclusion and Future Work	58
7.1 Summary	58
7.2 Contributions.....	58
7.3 Future Work.....	59
7.4 Closing.....	61
References	63
Appendix A: Core Implementation Scripts	67
Appendix B: Raw LLM Prompts and Context Data	78
Appendix C: Detailed Experimental Result	80
Appendix D: SYNERGIES Evaluation Questionnaires.....	84

List of Figures

Figure 1.1: Fragmented platform evaluation methods versus the unified AUREA architecture

Figure 2.1: Landscape of existing fragmented tools versus the integrated AUREA gap

Figure 3.1: Net Promoter Score Distribution

Figure 3.2: Task Completion Rate by Usability Test Step

Figure 3.3: Average response time and latency under increasing concurrent workloads

Figure 4.1: AUREA framework architecture overview

Figure 4.2: Sequence diagram of the AUREA execution pipeline

Figure 5.1: PagoPA response time over time

Figure 5.2: PagoPA Boolean Metrics Evaluation Dashboard (71% Pass Rate)

Figure 5.3: Just Eat response time over time

Figure 5.4: Just Eat Boolean Metrics Evaluation Dashboard (67% Pass Rate)

Figure 5.5: Execution time distribution across AUREA components (PagoPA audit)

List of Tables

Table 2.1: Summary of related state-of-the-art tools and frameworks

Table 3.1: KPIs of the SYNERGIES evaluation, methods, and results

Table 3.2: System responsiveness at 3000 concurrent users

Table 4.1: Requirements and the AUREA components that address them

Table 4.2: Coverage of the SYNERGIES KPIs by AUREA

Table 5.1: Sequential steps of the AUREA implementation pipeline

Table 5.2: Detailed implementation of deterministic compliance checks

Table 5.3: Source domain of the 21 Boolean compliance indicators

Table 5.4: Execution time distribution per operational step

Table 7.1: Limits of the current implementation and planned architectural changes

Table C.1: Run-by-run execution metrics for PagoPA

Table C.2: Run-by-run execution metrics for Just Eat UK

List of Abbreviations and Symbols

General Computing and AI

AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
LLM	Large Language Model
RAM	Random Access Memory

Web Technologies and Protocols

CSS	Cascading Style Sheets
DOM	Document Object Model
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JS	JavaScript
JSON	JavaScript Object Notation
REST	Representational State Transfer
SPA	Single Page Application
TLS	Transport Layer Security
XHR	XML Http Request

Evaluation Framework and Metrics

AIA	API Intent Assessment
CBAC	Cross-Browser and API Compatibility
CSAT	Customer Satisfaction Score
ER	Error Rate
KPI	Key Performance Indicator
NPS	Net Promoter Score
SE	Security Evaluation
SIA	Semantic Interoperability (Assessment)
SUS	System Usability Scale
TCA	Trust & Compliance Assurance
TCR	Task Completion Rate
UTI	Usability, Technicality, and Interoperability

Usability and Interface

ISO	International Organization for Standardization
UI	User Interface
UX	User Experience
WCAG	Web Content Accessibility Guidelines

Security and Cloud (AWS)

AWS	Amazon Web Services
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
GDPR	General Data Protection Regulation
KMS	Key Management Service
S3	Simple Storage Service
SES	Simple Email Service

Project-Specific

AUREA	Automated Unified Runtime Auditor
CCAM	Cooperative, Connected and Automated Mobility
DITEN	Department of Electrical, Electronic, Telecommunications
ODD	Operational Design Domain
RQ	Research Question
SCDB	Scenario Database

Metrics and Units

ms	Millisecond
%	Percentage
σ	Standard Deviation

Chapter 1 - Introduction

1.1 Background and Motivation

Digital platforms play a critical role in many modern domains, and users expect significantly more than basic computational functionality from them. Today, operators demand secure communication protocols, highly usable interfaces, mathematically reliable application programming interfaces, and strict compliance with complex legal frameworks such as the General Data Protection Regulation (GDPR)[1], [2]. Because many of these platforms handle highly sensitive personal information or safety relevant data streams, they must work reliably under stress, actively protect that data against external interception, and resist continuously evolving security vulnerabilities [1].

For this precise reason, platform evaluation can no longer exist as a single isolated audit conducted at the very end of the software development lifecycle. Khan et al. [3] explicitly note that organizations frequently treat cybersecurity and compliance as a mere afterthought, which guarantees they will keep facing severe security threats as a direct result. Quality and security checks have to continue dynamically during active development and persist long after public deployment [1]. Furthermore, these continuous audits have to cover much more than backend function: strict privacy compliance, technical robustness under load, and front-end human usability all matter equally to the final operational success of the system.

The domain of Cooperative, Connected and Automated Mobility (CCAM) [4] provides a clear and demanding example of this necessity. The SYNERGIES project, which directly motivates the research presented in this thesis, builds a massive, federated platform where academic researchers and industry partners share and query the complex scenario datasets utilized to validate CCAM systems [5]. While the SYNERGIES platform is not a physical vehicle control system itself, its distributed data actively feeds the downstream safety validation of autonomous algorithms, meaning its structural data quality matters immensely. The platform has to be technically sound at the architectural level, and human operators have to be able to use it efficiently without making critical configuration errors.

The second half of that operational requirement forms the precise analytical focus of this thesis. A software platform can be computationally fast and architecturally stable yet remain exceptionally hard to use for its intended audience. I collectively call these essential interaction aspects human factors. The ISO 9241 Part 11[6] specification officially defines usability as the exact

effectiveness, efficiency, and satisfaction with which specified users can reach specified goals within a strictly specified context of use; I utilize this foundational definition as summarized in. However, in this research, I theoretically extend this traditional definition with critical trust related qualities that actively shape how modern users perceive and interact with an automated system. I define these extended trust elements by evaluating how clear the system error messages are, how transparent the backend data handling is, and how visible the legal and privacy information remains during standard navigation. This specific theoretical extension is my own academic contribution and is not part of the original ISO definition. Crucially, these extended trust elements are among the primary aspects that the AUREA framework actively checks and measures [1].

In this thesis, the term human factors is used in a deliberately bounded sense. It does not cover the full spectrum of human-computer interaction, such as cognition, aesthetics, learnability, accessibility, or perceived satisfaction, which can only be assessed with real users. Instead, it refers to the subset of trust- and interaction-related qualities that leave observable traces at runtime: the visibility of privacy and legal information, the clarity and safety of error messages, structural responsiveness on mobile devices, and the behavior of the API as experienced by a client. AUREA measures these qualities through proxy indicators, not through direct observation of users. It therefore addresses a runtime-observable slice of human factors and does not claim a complete, 360-degree evaluation.

1.2 Problem Statement

Evaluation of digital platforms is still fragmented. Usability, technical performance, security, and legal compliance are usually assessed as separate domains, each with its own tools and methods. The UTI framework [7] is an attempt to unify them: it assesses Usability, Technicality, and Interoperability through 13 Key Performance Indicators (KPIs) that mix qualitative and quantitative measures. Even so, in practice many of its indicators are still measured with different instruments and judged by human experts [1], which takes time, adds subjectivity, and makes the evaluation hard to repeat for later releases. Manual expert reviews often suffer from cognitive fatigue and subtle cognitive biases, leading to inconsistent scoring across different evaluation sessions. Furthermore, modern agile software development cycles deploy new codes on a daily or weekly basis. A purely manual evaluation bottleneck fundamentally contradicts the speed of agile deployment, forcing development teams to either delay releases or skip critical human factor validations entirely.

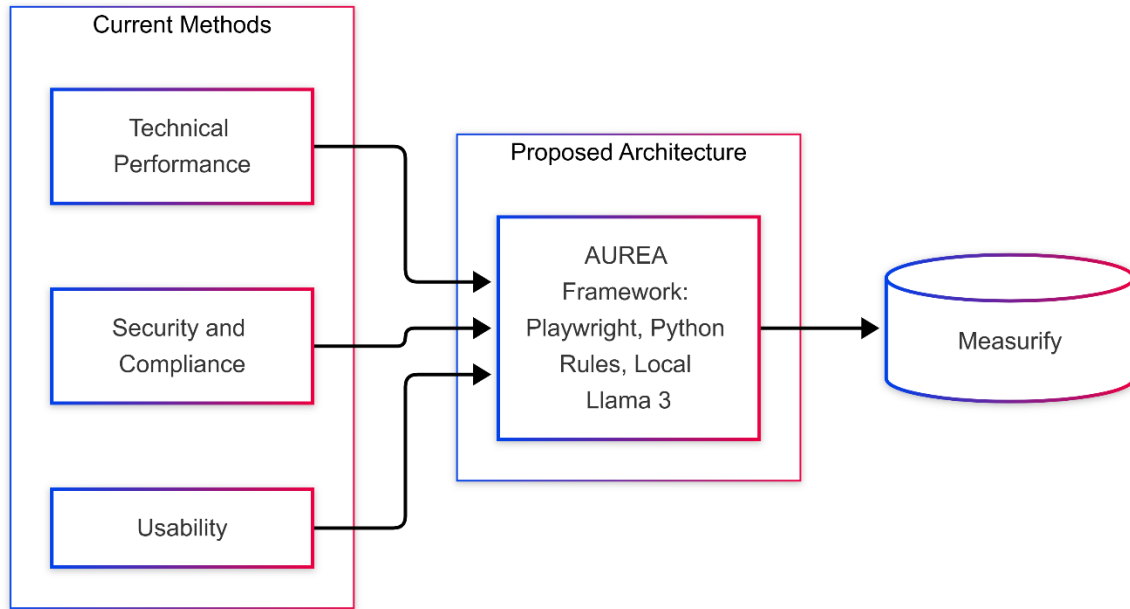


Figure 1.1: Fragmented platform evaluation methods versus the unified AUREA architecture

Automated tools do not close this gap, because most of them cover a single aspect. Cookiescanner detects and evaluates GDPR consent notices [8]. PRISMe uses an LLM to assess privacy policies [9]. AccessGuru [10] and ACCESS [11] combine browser automation with LLMs to detect and correct accessibility violations. AccessGuru is close to the approach of this thesis, since it pairs a deterministic tool (Axe-Playwright) with an LLM for the semantic cases, but it stays inside accessibility. The integration of API behavior, security, legal compliance, and human-factor checks in one automated runtime is still largely unexplored.

Automation also has a known limit. Vigo et al. compared six accessibility tools against expert evaluation and found that even the best one covered at most half of the WCAG success criteria [12]. They conclude that relying only on automated tests has negative effects. I take this seriously in the design of this work: automation should support human evaluation, not replace it.

1.3 Motivation

The motivation for this thesis comes from my direct professional involvement in the SYNERGIES project (Horizon Europe, GA No. 101146542). SYNERGIES developed a highly complex, federated European data ecosystem designed specifically for sharing, processing, and validating massive scenario datasets for CCAM systems. Working as an intern on the project at the University of Genoa from 2024 to 2025, under the supervision of Prof. Riccardo Berta, I carried out

the hands on evaluation work of the first major platform release. My core responsibilities focused on designing Key Performance Indicator evaluation frameworks and executing data driven scenarios to test the platform behavior under real world conditions. The formal results of this extensive evaluation phase are thoroughly documented in Deliverable D8.2 [5].

The evaluation used 13 KPIs in three dimensions, matching the structure of the UTI framework [7]: usability, technical performance, and interoperability. Usability was measured with supervised tests on 24 participants (17 from the consortium and 7 external) and with CSAT, SUS, and NPS questionnaires.

Technical performance evaluation required heavy reliance on specialized industrial software. I utilized Apache JMeter [13] to conduct extensive load testing. This open source Java application is specifically designed to simulate massive concurrent user traffic, allowing the evaluation team to accurately measure network bottlenecks, server response times, and overall backend resilience under extreme computational stress. Simultaneously, the structural quality of the platform was evaluated using SonarQube [14]. SonarQube functions as an advanced continuous inspection platform that performs automated static code analysis to detect hidden programming bugs, latent security vulnerabilities, and architectural inefficiencies, ensuring the codebase maintains high maintainability. Finally, security, compliance, cross-browser compatibility, and API interoperability were comprehensively assessed by three experts utilizing manual checklists.

The results showed a clear gap. On the technical side the platform performed very well: with 3000 concurrent users, the average response time was 112 ms and the error rate was 0%. Apache JMeter [13] validated the robustness of the network throughput. On the user side, the picture was different. CSAT was 67.5% against a target of 85%, SUS was 59.9 against a target of 75, NPS was -27.7 against a target of 60, and only 14% of participants completed the selection of Operational Design Domain (ODD) parameters on their own [5]. Technical strength alone did not make the platform easy to use.

The evaluation process also had limits that I saw directly. Each dimension needed a different tool or method. Several KPIs depended on expert judgment. Some checklist items could not be assessed at first because the evaluators had no access to the needed information, and they were resolved later through code review. Problems such as unclear terminology and technical details in error messages were found late and logged by hand in a shared spreadsheet [5].

From this experience I drew two conclusions. Human-factor aspects are the ones that most affect adoption, and they are also the ones least supported by

automation. And part of what I did by hand, such as checking responsiveness, cookie security, API behavior, error messages, and privacy transparency, could in principle be automated.

1.4 Proposed Solution and Research Questions

To address these limits, this thesis presents the Automated Unified Runtime Auditor (AUREA), a framework that combines Playwright-based browser interaction[15], deterministic Python validation, and semantic reasoning with a locally hosted LLM (Llama 3 through Ollama) [16], [17] in one session. Results are stored and shown through Measurify [18], [19], [20]. Because the LLM runs locally, sensitive content such as privacy policies and API payloads stays on the machine. AUREA was first presented in [1].

AUREA is meant to complement existing methods, not replace them. It does not cover load testing, static code analysis, or user studies with questionnaires, which remain necessary to measure real user perception. It covers those human-factor aspects that can be checked automatically at runtime.

The thesis answers four research questions:

- **RQ1.** How is the evaluation of human-factor aspects carried out in a real project such as SYNERGIES, and what limits emerge?
- **RQ2.** Can browser automation, deterministic validation, and a local LLM be combined to automate part of this evaluation in one workflow?
- **RQ3.** Does the approach work on platforms with different technologies and content, without site-specific changes?
- **RQ4.** What are its limits, in terms of LLM reliability, privacy, execution time, and coverage of human factors?

The main contributions are:

1. An analysis of the SYNERGIES evaluation and its limits.
2. The requirements and design of AUREA.
3. An evaluation of AUREA on two live platforms, PagoPA [21] and Just Eat [22], with 25 runs each and 21 Boolean metrics.
4. A discussion of what automation can and cannot do.

The idea of AUREA was mine, and I carried out its design, implementation, and experimental evaluation. The framework was presented at the 11th South-East Europe Design Automation, Computer Engineering, Computer Networks and

Social Media Conference (SEEDA-CECNSM 2026), held in Thessaloniki, Greece, on 18–20 September 2026 [1], and was co-authored with my supervisor Prof. Riccardo Berta and colleagues at the University of Genoa. This thesis is built on that paper and extends it with a detailed analysis of the SYNERGIES evaluation that motivated the work, a full derivation of requirements from the limits found there, and a deeper discussion of the limitations of the approach. The Measurify platform used for storage and visualization comes from earlier work of the same research group.

1.5 Scope of the Thesis

This thesis covers the evaluation of publicly accessible pages and the application programming interface calls that those specific pages generate. Authenticated areas located behind login screens fall outside the current operational scope, because the framework currently lacks automated session management; the foundational paper [1] lists this capability as required future work. Furthermore, the automated approach is not designed to replace volumetric load testing, static source code analysis, or empirical user studies. It covers only the specific subset of compliance and structural checks that can be executed at runtime from outside the platform boundary (detailed in Section 4.4).

The experimental validation utilized two live production websites: the public home pages of PagoPA (www.pagopa.gov.it) [21] and Just Eat UK (www.just-eat.co.uk) [22]. The script executed 25 runs per platform on a standard consumer laptop equipped with an Intel Core i3 processor and 8 gigabytes of RAM. Consequently, the reported execution times depend heavily on this specific hardware configuration [1]. The literature describes PagoPA as a payment platform of the Italian public administration featuring linear transaction flows and formal legal documentation, while Just Eat serves as a commercial single page application characterized by frequent front end updates [1]. It is critical to clarify that the audit itself did not navigate through the actual payment processing or food ordering flows. The script evaluated the initial home page and one automatically selected background endpoint per run. Therefore, the extracted results describe only those isolated pages and do not represent the complete platform services. Because it covers only two targets and the findings lack formal validation by domain experts, this evaluation must be understood as an experimental pilot [1].

To maintain ethical boundaries, the automated tests utilized benign requests, specifically deploying a modified user agent string, transmitting duplicate requests, and sending malformed parameters designed to provoke standard error codes. During these procedures, no sensitive data was accessed or

retained [1]. Any future extension of this framework, particularly regarding authenticated testing into areas behind a login authentication barrier, would require the explicit legal permission of the platform owner.

The inclusion of the word industrial in the thesis title requires a precise academic comment. The SYNERGIES platform represents a Horizon Europe research project situated within the automotive domain, officially coordinated by IDIADA Automotive Technology SA [5]. Conversely, PagoPA and Just Eat are not industrial platforms in the strict definition of the term. I selected these specific public targets because they differ significantly in their fundamental type and content structure. This selection proves whether the framework can run successfully on different sites without requiring manual code changes, but it does not show that the architecture works on every possible kind of digital platform.

A complete inventory detailing the limits of the Python implementation is provided in Section 5.8, and a comprehensive discussion regarding the theoretical threats to the validity of the final results is located in Section 6.3.

1.6 Thesis Structure

The rest of the thesis is organized as follows:

Chapter 2 reviews the background on usability and privacy evaluation, LLMs in software testing, and related frameworks, and states the gap.

Chapter 3 analyzes the SYNERGIES evaluation and derives the requirements.

Chapter 4 presents the architecture of AUREA.

Chapter 5 describes the implementation, the experimental setup, and the results on PagoPA and Just Eat UK, and lists the limits of the implementation.

Chapter 6 discusses the results and the threats to validity.

Chapter 7 concludes and outlines future work.

Chapter 2 - Literature Review and Related Work

2.1 Overview of Digital Platform Evaluation

Usability is usually defined through three components: how effectively, how efficiently, and how satisfactorily specified users reach their goals in a specified context of use [6]. In the SYNERGIES evaluation I used this view in practice. Effectiveness and efficiency were observed through task completion and errors during supervised sessions, and satisfaction was collected with the CSAT, SUS, and NPS questionnaires [5]. These psychometric instruments offer a structured approach to quantifying subjective user experiences. The Customer Satisfaction Score measures immediate task satisfaction, the System Usability Scale evaluates overall platform usability across ten standardized items, and the Net Promoter Score gauges long term user loyalty. However, administering these tools requires significant logistical overhead and participant coordination. The SUS result is a good example of how these instruments are read: the platform scored 59.9, and the evaluation report places anything below 68 in the "below average" range. Such results are useful, but they need participants, supervision, and time, and with 24 participants they are better suited to finding major problems than to producing stable benchmarks [5].

2.2 Usability and Privacy Auditing Principles

Human factors also include trust-related qualities that questionnaires capture poorly. Whether a privacy policy is visible and understandable, whether data collection is described honestly, and whether error messages avoid exposing technical details are all judged by experts with checklists [5]. In SYNERGIES, points were lost in the security evaluation precisely because some error messages revealed implementation details [5]. These are the kind of checks that depend on reading text in context, which makes them hard to automate with fixed rules. Evaluating these nuanced elements often falls outside the scope of traditional unit testing or automated performance metrics. A human expert must contextually interpret whether a legal document is visually prominent or if an interface naturally guides the user toward secure choices without overwhelming them with cognitive load.

Most automated tools address one aspect of a platform. For legal compliance, Cookiescanner detects and evaluates GDPR consent notices on websites [8]. For privacy policies, PRISMe uses an LLM to help users understand what a policy says, and its authors also report concerns about hallucinations in the model's output [9]. Such hallucinations pose a severe risk in legal or compliance

auditing, where an artificial intelligence fabricating a clause could lead to false positive compliance scores. Therefore, deploying generative models for evaluation requires strict prompt engineering to force deterministic and verifiable outputs.

2.3 Large Language Models in Software Testing

Accessibility is the most active area. AccessGuru [10] and ACCESS [11] pair browser automation with LLMs to detect and correct HTML violations. López-Gil and Pereira showed that LLM-based scripts can automate checks that used to need manual review [23]. Gu et al. proposed the AAA framework, in which an LLM copilot samples DOM structures obtained by automated crawling [24]. All of these work at the front end.

Automation has limits that the accessibility literature documents well. Vigo et al. compared six tools against expert evaluation and found that even the best one covered at most half of the WCAG success criteria, which led them to warn against relying on automated tests alone [12]. I take this seriously in the design of this work: automation should support human evaluation, not replace it.

The accessibility frameworks above share a pattern. A deterministic step collects or checks what can be measured directly, such as the DOM or rule violations, and an LLM handles what fixed rules cannot, such as whether a text is clear or an element is labelled meaningfully. The same pattern has been explored much less outside accessibility. In particular, combining traditional API and security testing with LLM reasoning, for example to judge whether a server error message is clear or leaks sensitive information, remains largely unexplored. A second open point is privacy. Privacy policies, API payloads, and interaction data are sensitive, which is an argument for running the LLM locally instead of sending them to an external service. Executing the language model strictly locally ensures that all sensitive intellectual property remains entirely within the secure development environment.

2.4 SYNERGIES Project Context

I met this gap in practice during the SYNERGIES project, whose first release I helped evaluate. SYNERGIES developed a federated European platform for sharing scenario datasets for CCAM systems. The platform was technically strong with 112 ms average response time with 3000 concurrent users, but the usability results were weak, and only 14 percent of participants completed the

selection of Operational Design Domain parameters on their own. The evaluation needed JMeter and SonarQube for technical performance, questionnaires for user perception, and expert checklists for security, compliance, and interoperability. This fragmented approach highlights a significant industrial pain point. Navigating between load testing software, static code analyzers, manual spreadsheets, and legal checklists creates massive friction for quality assurance teams. Consolidating the runtime aspects of these distinct evaluations into a single automated pipeline represents a major opportunity to optimize the software engineering lifecycle.

2.5 Integrated Frameworks and the Gap

The UTI framework is the closest attempt at integration. It evaluates Usability, Technicality, and Interoperability within a single KPI-based assessment. As discussed in, a limitation is that all indicators are assessed by human experts, which increases cost and evaluation time and introduces subjectivity [7]. This reliance on human experts creates a severe bottleneck in modern agile software development environments. Table 2.1 summarizes the related state-of-the-art tools discussed in this chapter, highlighting their primary focus and limitations compared to the unified approach of this thesis.

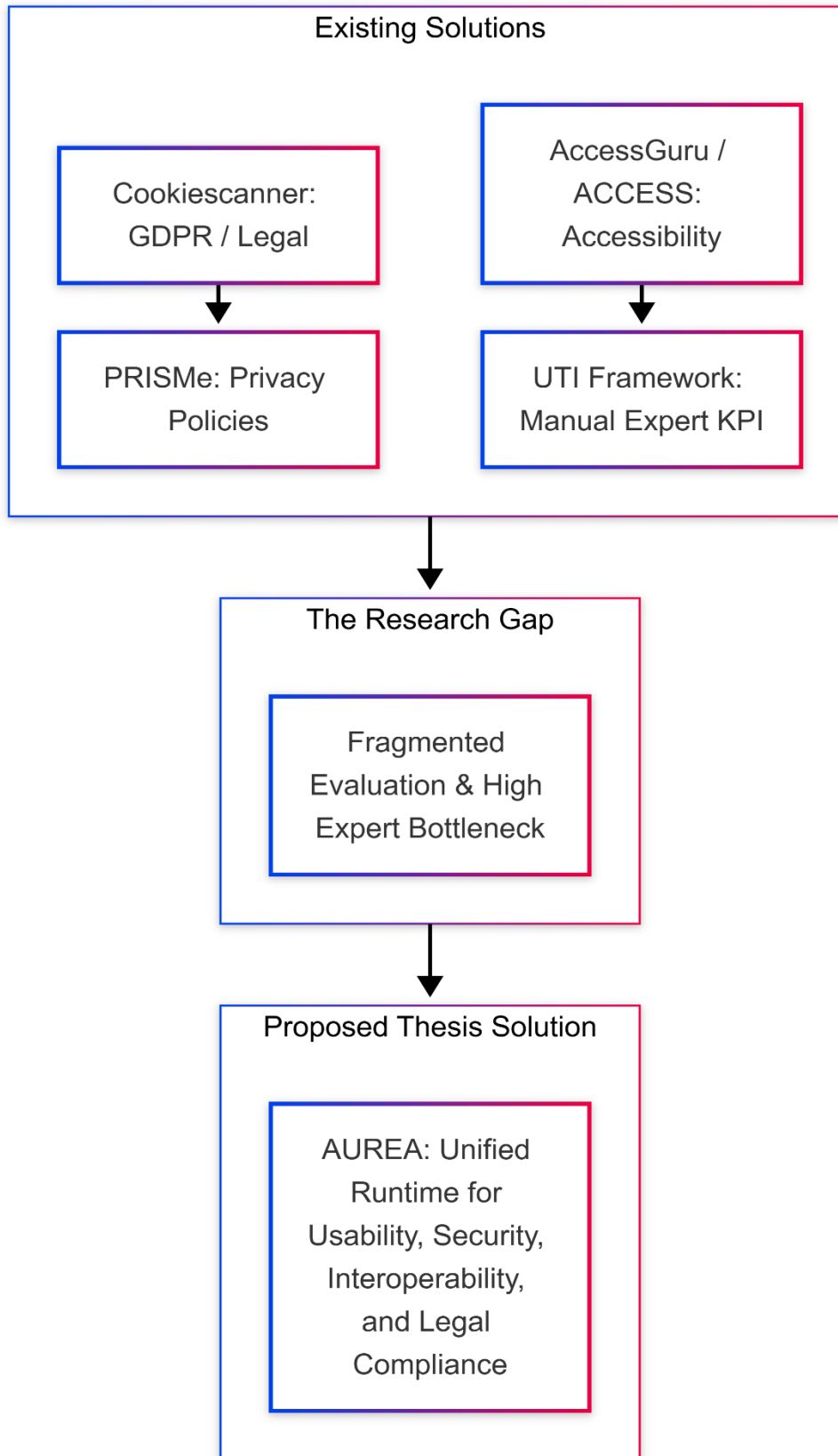


Figure 2.1: Landscape of existing fragmented tools versus the integrated AUREA gap

Work	Main focus	Method	Limitation for this thesis
Cookiescanner [8]	GDPR consent notices	Automated scanning and DOM analysis	Single aspect
PRISMe [9]	Privacy policies and transparency	LLM-based assessment, user study	Single aspect; hallucination concerns
AccessGuru [10], ACCESS [11]	Accessibility violations	Browser automation and LLM evaluation	Front-end focus on
López-Gil and Pereira [23]	Accessibility success criteria	Web accessibility testing tools and LLM	Front-end focus on
Gu et al. (AAA) [24]	Large-scale accessibility auditing	Crawling and LLM copilot	Front-end focus on
UTI Framework [7]	Usability, technicality, interoperability	KPI-based framework	Indicators assessed exclusively by human experts
AUREA (This thesis)	Usability, security, interoperability, legal compliance	Playwright, Python scripts, Local Llama 3	Covers only public pages at runtime

Table 2.1: Summary of related state-of-the-art tools and frameworks

What is missing is a single runtime that covers API behavior, security, legal compliance, and human-factor checks together, using deterministic validation where possible and a local LLM where context is needed. This is the gap that AUREA addresses.

2.6 The SYNERGIES Context

I met this gap in practice during the SYNERGIES project, whose first release I helped evaluate. SYNERGIES developed a federated European platform for sharing scenario datasets for CCAM systems [5]. The platform was technically strong (112 ms average response time with 3000 concurrent users), but the usability results were weak, and only 14% of participants completed the selection

of Operational Design Domain parameters on their own [5]. The evaluation needed JMeter and SonarQube for technical performance, questionnaires for user perception, and expert checklists for security, compliance, and interoperability [5]. This fragmented approach highlights a significant industrial pain point. Navigating between load testing software, static code analyzers, manual spreadsheets, and legal checklists creates massive friction for quality assurance teams. Consolidating the runtime aspects of these distinct evaluations into a single automated pipeline represents a major opportunity to optimize the software engineering lifecycle. Chapter 3 analyzes this evaluation in detail.

Chapter 3 - Problem Identification and Case Study

3.1 The SYNERGIES Platform and Its Evaluation

SYNERGIES developed a federated European platform for sharing and validating scenario datasets for CCAM systems [5]. These datasets are vastly different from traditional relational databases. They encompass massive arrays of autonomous vehicle sensor telemetry, complex weather condition simulations, and highly specific edge case driving scenarios utilized to train artificial intelligence driving models. Managing and validating such intricate data structures requires an exceptionally intuitive user interface to prevent operator errors during dataset selection and platform configuration. Its first release was evaluated within WP8 of the project, using the KPI framework of Deliverable D8.1 [5]. The framework has three dimensions (usability, technical performance, and interoperability) and 13 KPIs, which matches the structure of the UTI framework [7]. I took part in this evaluation as described in Section 1.3.

Twenty-four participants took part in the usability sessions: 17 consortium members and 7 external users [5]. D8.2 itself notes that this is a limitation, because consortium members know the platform's concepts and terminology better than newcomers [5]. Each KPI was measured with a different method. Table 3.1 lists them with the results.

Dimension	KPI	Method	Result	Target
Usability	CSAT	Questionnaire (24 participants)	67.5%	85%
Usability	SUS	Questionnaire (24 participants)	59.9	75
Usability	NPS	Questionnaire (24 respondents)	-27.7	60
Usability	TCR / ER	Supervised usability test (14 participants)	ODD parameters: 14%	>85%
Usability	TCA	Three experts, checklist	88.33%, then 96.50% after code review	90%

Usability	CBAC	Three experts, checklist	82.54%	90%
Technical	System responsiveness	Apache JMeter	112 ms average	<1000 ms
Technical	Extensibility	SonarQube	Code coverage 7.3% (back-end), 9.0% (front-end)	>50%
Technical	Security	Three experts, checklist	68.25-92.15%	-
Technical	Resilience	Not applied	Not evaluated	Not applicable
Interoperability	API intent (AIA)	Three experts, checklist	100%	85%
Interoperability	Semantic interoperability (SIA)	Three experts, checklist	83.33%	70%

Table 3.1: KPIs of the SYNERGIES evaluation, methods, and results (data from [5])

3.2 Usability Results

The usability KPIs were all below target. CSAT was 67.5 percent against 85 percent. A modified six item version of the System Usability Scale was utilized to evaluate the platform [5], resulting in a score of 59.9 against 75, which D8.2 places in the "below average" range. Furthermore, NPS was negative 27.7 against 60 (2 promoters, 9 passives, and 7 detractors) [[5], Tables 1, 4, 5].

■ Passives (7-8) ■ Detractors (0-6) ■ Promoters (9-10)

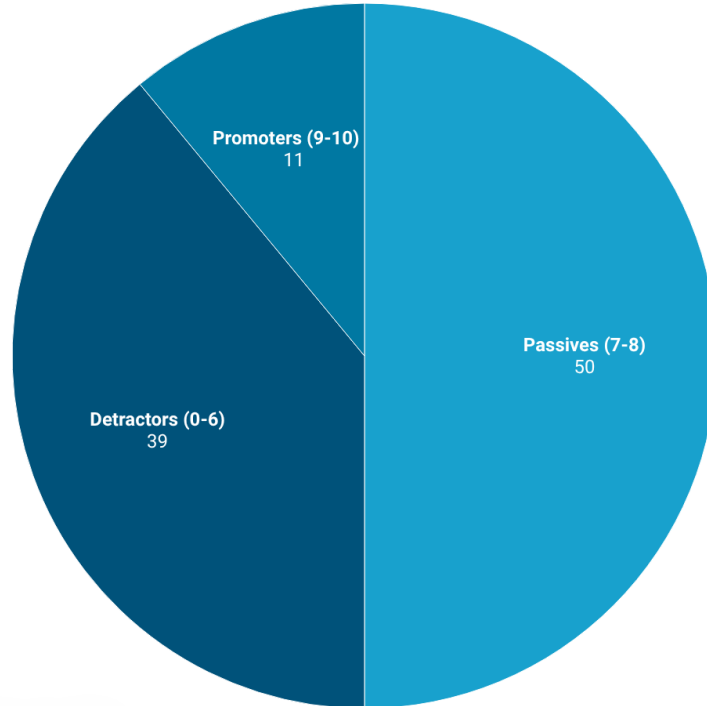


Figure 3.1: Net Promoter Score Distribution

NPS was the largest gap of all KPIs. D8.2 also points out that the target of 60 is ambitious for a first release and that expert participants may judge against mature commercial tools [5].

While 24 participants completed the evaluation questionnaires, the supervised task execution and completion monitoring were conducted with a smaller subset of 14 users.

The task completion data show where the problems were. Finding the dataspace, finding the "+" button, selecting the dataset, selecting the ODD, and saving the query were completed by 100% of participants. Executing the query reached 92.8%, and downloading the package and changing the query name reached 85.7%. Selecting the ODD parameters was completed independently by only 14% [[5], Table 7].

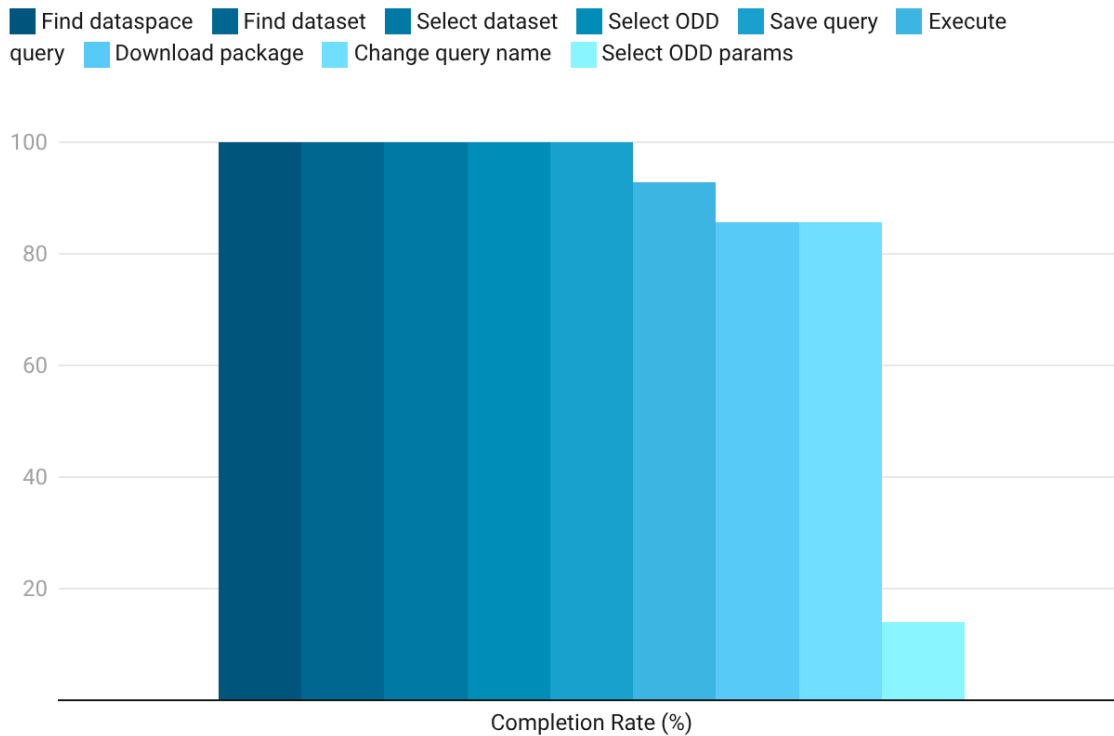


Figure 3.2: Task Completion Rate by Usability Test Step

The Operational Design Domain defines the strict environmental and systemic conditions under which an automated driving system is designed to function safely. This encompasses a multidimensional matrix of variables including illumination levels, precise weather conditions, road geometry constraints, and dynamic traffic density. The platform interface presented these highly complex parameters using raw database nomenclature rather than plain language, technical terminology that was unfamiliar to most users. The observers noted that the parameters were presented with technical terminology that most users did not know, and that several participants tried to use a search function that does not exist in the interface [5].

The questionnaires point to the same places. Ten participants reported that values and naming were unclear or inconsistent, six found defining queries confusing, and five found adding ODD criteria confusing and time-consuming [[5], Table 3]. So, the problem was not the platform in general, but a few specific steps, mainly query definition and ODD selection. Since most of the platform worked well and these steps did not, backend quality alone did not make the platform usable.

3.3 Technical Performance and Interoperability

The technical results were much better. The JMeter tests used two types of operation: queries (retrieval of scenario database metadata) and mutations (scenario execution), with 1000, 3000, and 5000 concurrent users [5]. The error rate was 0% in all three cases. The Apache JMeter testing suite achieved this extreme load simulation by utilizing distributed thread groups to generate massive volumes of synthetic HTTP requests. The simulation explicitly employed a 15 ms ramp-up period to prevent artificial network bottlenecks, accurately mimicking the organic traffic patterns of a massive distributed engineering team querying the scenario database simultaneously across the European continent. D8.2 compares the results with the targets at 3000 users (Table 3.2).

Scenario	Concurrent users	Avg. response time (ms)	Error rate	Latency (ms)
Query	3000	96	0%	95
Mutation	3000	127	0%	127
Total	6000	112	0%	111

Table 3.2: System responsiveness at 3000 concurrent users (from [5], Table 10).

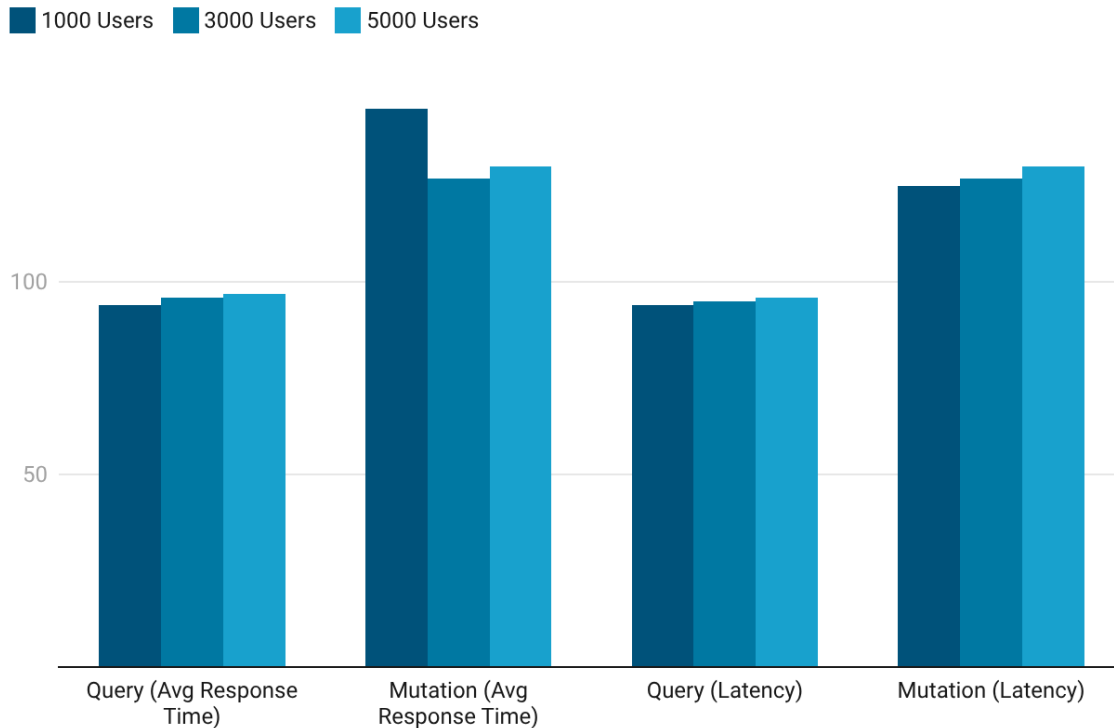


Figure 3.3: Average response time and latency under increasing concurrent workloads

The average response time of 112 ms is far below the 1000 ms target, and the latency of 111 ms is below the 200 ms target [8, Table 12]. The interoperability experts gave the API a score of 100% for intent and 83.33% for semantic interoperability. The deduction comes from API versioning (for example /v1/ tags), which is still partly missing in some modules [5].

The technical picture is not uniformly positive, though. The SonarQube analysis found code coverage of only 7.3% for the backend and 9.0% for the frontend against a target of over 50%. The average cyclomatic complexity per backend function was 21 against a target below 10 [[5], Tables 13, 14]. Cyclomatic complexity is a critical software metric that formally measures the total number of linearly independent paths through a program source code. A severe score of 21 indicates deeply nested conditional logic and overly dense control flow graphs. Functions exhibiting such high complexity are incredibly difficult to unit test comprehensively, are heavily prone to regression bugs during agile updates, and demand immense cognitive effort from new developers attempting to read or maintain the underlying architectural logic. So performance was excellent, while maintainability needed work.

3.4 Expert-Based Assessment

Five KPIs (TCA, CBAC, security, AIA, and SIA) were assessed by three experts using checklists [5]. This is where the process showed its limits most clearly.

For Trust & Compliance Assurance (TCA), the experts could not see the backend, so the items on log archiving and breach notification were marked "inaccessible" and left out of the score, which was 88.33%. A code review later confirmed that both mechanisms exist (log archiving through Amazon S3 Glacier and breach notification through Amazon SES), and the score was updated to 96.50% [5]. The result therefore depended on how much the experts could see.

The security evaluation was performed by three experts and received a score ranging from 68.25 percent to 92.15 percent [5]. The experts confirmed that HTTPS and role-based access were correctly enforced. They deducted points because some error messages exposed technical details [5]. Specifically, some technical details are revealed in error messages.

This is the same kind of check that AUREA later automates. Cross browser compatibility (CBAC) reached 82.54 percent, with consistent layouts in Chrome, Firefox, Edge, and Safari, but with console errors in some older browser versions [5]. The resilience KPI could not be evaluated because the fault injection infrastructure was not yet available [5].

The issues found in testing were collected by hand in a shared spreadsheet. It contained 15 entries: 5 usability issues, 4 interface issues, 4 bugs, and 2 login issues. Fourteen of them still had the status "waiting for check" at the time of D8.2 [[5], Table 8].

3.5 Limits of the Evaluation Process

Looking at the whole evaluation, the platform results are only half of the story. The process also had limits, which are the starting point of this thesis.

1. **Several tools and methods.** Each dimension needed its own tool or procedure, and the results had to be combined by hand.
2. **Dependence on expert judgment.** Five of 13 KPIs relied on checklists filled in by experts, which is subjective and hard to repeat for the next release. As D8.2 shows for TCA, the result also depends on what the experts can see.
3. **Late and manual reporting.** Problems such as unclear terminology or technical details in error messages were found late and logged by hand in a shared spreadsheet.

4. **Manual setup.** Even a technical step such as code analysis needed manual preparation. To run SonarQube, I had to start a local instance in Docker and run the scanner from the terminal, with an authentication token, because otherwise the scanner failed with authorization errors:

Powershell

```
docker run -d --name sonarqube -p 9000:9000 sonarqube:community  
npx sonarqube-scanner "-Dsonar.host.url=http://localhost:9000" "-  
Dsonar.token=<TOKEN>" "-Dsonar.projectKey=my-ui-project"
```

Missing coverage reports were another source of errors at this stage. None of this is difficult, but it takes time and makes the evaluation harder to reproduce. This manual orchestration completely breaks the fundamental concept of continuous integration. A quality assurance engineer must manually provision the Docker container environment, precisely configure the local network routing ports, securely inject the authentication tokens via the command line interface, and execute the runtime scripts sequentially. This highly disjointed operational workflow virtually guarantees that comprehensive evaluation remains a sporadic, isolated event rather than a continuous developmental guarantee seamlessly integrated into the software deployment pipeline.

From these limits I derived the requirements for AUREA (Chapter 4). It should:

- **R1.** collect the data for several dimensions in one session;
- **R2.** check automatically what can be measured directly (responsiveness, cookie flags, API format, error codes);
- **R3.** use semantic reasoning for what depends on context (privacy text, error message clarity, naming);
- **R4.** keep the results in one structure;
- **R5.** be repeatable for later releases;
- **R6.** keep sensitive data on the local machine.

AUREA does not aim to replace the load testing, the code analysis, or the questionnaires in Table 3.1, because these measure things that cannot be checked from outside a running page. It covers the part of the expert-based checks that can be automated.

Chapter 4 - The AUREA Framework

4.1 Design Principles and Requirements

The limits found in Chapter 3 led to the six requirements R1 to R6. AUREA [1] is designed to meet them in one pipeline that audits the usability related, security, compliance, and interoperability aspects of a web platform during a single browser session. It is meant to complement the load testing, code analysis, and user studies of Table 3.1, not to replace them.

R1 (one session). Playwright [15] drives a real browser through the platform and captures, in one run, the rendered DOM, background API requests and responses (XHR and Fetch), HTTP status codes and headers, cookie events, and network metadata [1]. state is absolutely critical for scientific validity. If the framework utilized separate disconnected sessions for network analysis and user interface rendering, the dynamic state of the application could easily mutate between tests. This would render the final compliance metrics completely inconsistent, as the security evaluation might target a different interface state than the usability evaluation.

R2 (direct measurement). Python rules check what can be verified without an LLM: the viewport meta tag, the HttpOnly and Secure cookie flags, bot protection (a request with the User Agent "BadBot/1.0" should receive a 403), API idempotency and JSON validity (duplicate requests), and the response to malformed inputs (400, 404, and 500 errors) [1].

R3 (semantic reasoning). Llama 3[17], run through Ollama [16], handles what fixed rules cannot: GDPR transparency, privacy policy content, API naming consistency, and the clarity of API error messages [1]. Prompts are constrained, the temperature is low, and the model answers in a binary format (0 or 1) that is then parsed by deterministic pattern matching [10].

R4 (one result structure). The outputs of both analysis steps are stored in a common structure and sent to Measurify [1], [19].

R5 (repeatability). The pipeline is scripted, so it can be run again for later releases. LLM output can still vary between runs, which is why the prompts are constrained and the answers binary. Some steps may also need a person, for example to solve a CAPTCHA or close an overlay [1]. I discuss the remaining variability in Chapter 6.

R6 (local data). The LLM runs locally, so the content passed to it (privacy policies, API payloads, interaction data) does not leave the machine [1].

Requirement	Design decision	Component (as in [1], Table II)
R1	One browser session collects all runtime data	Runtime Interaction and Data Acquisition (Playwright)
R2	Rule based checks without LLM calls	Deterministic Validation (Python scripts)
R3	LLM for contextual checks, constrained binary output	Semantic Reasoning (Llama 3 via Ollama)
R4	Common result structure	Results Reporting
R5	Scripted pipeline, low temperature prompts	All steps
R6	Local LLM execution	Ollama

Table 4.1: Requirements and the AUREA components that address them.

4.2 Architecture

AUREA works as a five step workflow [1]: (1) runtime interaction and data acquisition with Playwright, (2) deterministic validation with Python scripts, (3) semantic reasoning with Llama 3, (4) results reporting, and (5) visualization in Measurify.

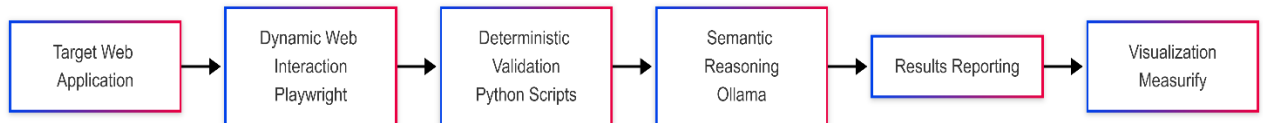


Figure 4.1: AUREA framework architecture overview

Steps 1 to 3 form the acquisition and analysis part, and steps 4 and 5 the reporting and storage part. The steps run one after the other, not in parallel. To formally illustrate this synchronous execution model, Figure 4.2 presents a comprehensive Unified Modeling Language sequence diagram. It details the exact chronological data flow starting from the initial Playwright browser navigation, traversing through the Python deterministic validation engine, proceeding into the local Ollama inference server, and finally transmitting the aggregated payload out to the remote Measurify database.

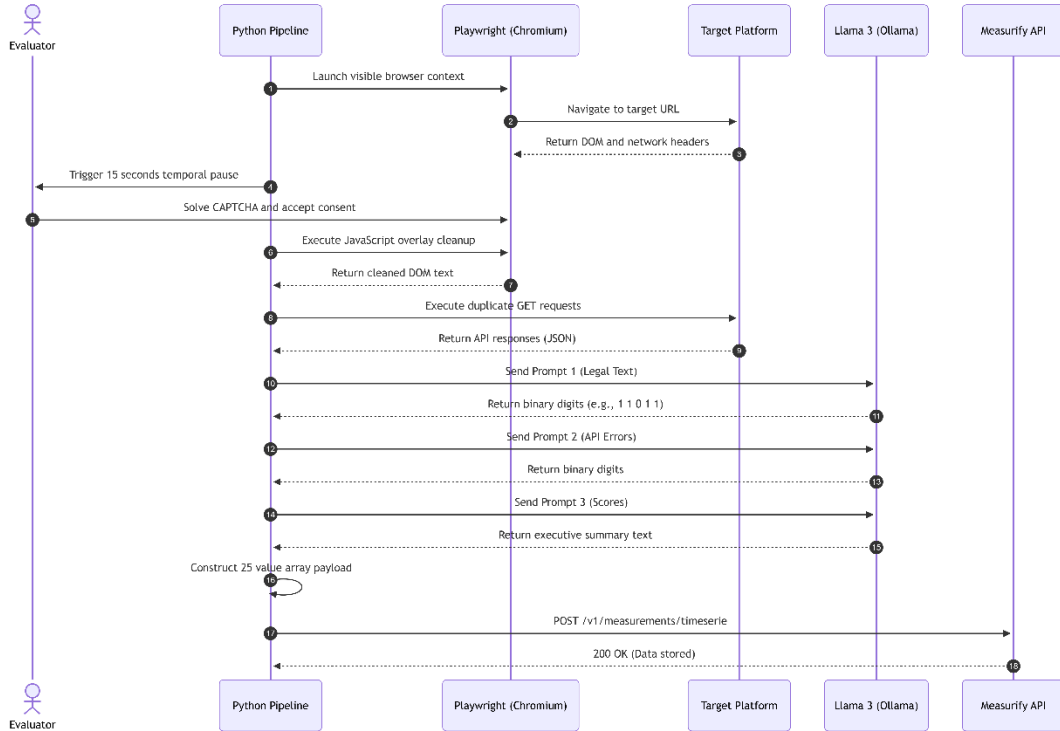


Figure 4.2: Sequence diagram of the AUREA execution pipeline

Steps 2 and 3 both work on the data collected in step 1, and step 3 also uses the error responses recorded in step 2, which it checks for exposed technical information [10]. Step 1 renders the page fully, so it also sees single page application content and API calls that are generated after the first load, which static HTML parsing would miss [1].

The split between rules and LLM has a practical reason. Rules are fast and exact, so they take the checks that can be measured directly, and the LLM is called only where context matters. This matters because the LLM step dominates the run time: on the laptop used in the experiments, step 3 took 564 s of the 602 s needed for the PagoPA audit [[1], Table III].

4.3 Output and Data Model

AUREA produces three kinds of output [1]: quantitative metrics (for example API response time and execution overhead), binary indicators for rule based and semantic checks, and short textual summaries written by the LLM. The binary indicators are 21 Boolean metrics covering, among others, HTTPS, privacy policy, GDPR compliance, data transparency, API idempotency, JSON format, error handling, versioning, mobile responsiveness, bot testing, and cookie flags [[1], Figs. 7 and 8].

Measurify stores and displays the results. It has a database, an API layer, and a graphical interface called KPI Manager [18], [19], [20]. Listing 4.1 shows an excerpt of the data payload generated by the framework. The values correspond to the PagoPA audit, where the framework structures the extracted metrics and the Llama 3 executive summary into a unified time series array before transmission. This highly compressed array structure is deliberately designed for extreme network efficiency. Because Measurify originally operates as an Internet of Things database optimized for lightweight continuous sensor telemetry, transmitting complex nested objects would create unnecessary computational overhead. The fixed length array ensures rapid database indexing and immediate visualization rendering on the KPI Manager dashboard.

```
{
  "timestamp": "2026-10-01T14:11:00.000Z",
  "device": "PagoPA_Device_v4",
  "values": [
    1,
    320,
    1500,
    1,
    1,
    1, 1, 0, 1, 1,
    1, 1, 0, 1, 1,
    457, 0, 1, 1, 1,
    1, 1, 1, 0,
    "- Legal and Compliance: Pass - Explicit mentions of GDPR and Privacy Policy found.\n- API Quality and Errors: Fail - Error responses expose internal technical details.\n- Bot Security: Pass - The bot request was successfully blocked."
  ]
}
```

4.4 What AUREA Covers and What It Does Not

Table 4.2 compares the KPIs of the SYNERGIES evaluation (Table 3.1) with the checks that AUREA performs. It shows that AUREA covers part of the expert based checks and none of the measurements that need load, source code, or users.

SYNERGIES KPI [5]	Related AUREA check [1]	Coverage
TCA (privacy, GDPR)	Prompts on privacy policy, consent, opt out, and data transparency	Partial
CBAC	Viewport tag for mobile responsiveness	Not measured (fixed to 1 in v4.3)
Security (SE)	Cookie flags, bot detection, error messages after malformed requests	Partial (no vulnerability scanning)
AIA, SIA	Idempotency, JSON validity, naming, versioning	Partial
System responsiveness	Response time of single requests	Not covered (no load test; JMeter remains necessary)
Extensibility	None	Not covered (needs source code; SonarQube)
CSAT, SUS, NPS, TCR / ER	None	Not covered (need users)

Table 4.2: Coverage of the SYNERGIES KPIs by AUREA.

One finding in particular is not addressed by the published version of AUREA: the terminology problem in the ODD selection of SYNERGIES (Section 3.2). AUREA semantic step checks legal text and API quality, not whether interface labels are understandable for non experts. Extending the LLM step to interface labels is a natural next step, which I discuss in Chapter 7. This is in line with the limit noted by Vigo et al. for automated tools [12]: automation covers part of what experts check, not all of it.

AUREA has further design limits [1]. It works on public pages only. It depends on the context size of the local LLM, so very long legal documents need special handling. And it needs a person for some interactions, such as CAPTCHAs.

4.5 Data Acquisition and Human in the Loop Integration

The data extraction module of AUREA relies on Playwright to drive a visible Chromium browser instance. Unlike traditional headless scraping tools, this architecture intentionally renders the browser interface to allow a human in the loop interaction model. Modern enterprise platforms increasingly deploy

aggressive Web Application Firewalls and heuristic behavioral analysis algorithms specifically designed to detect and immediately block automated scraping scripts. Attempting to programmatically bypass these advanced security perimeters is not only computationally expensive but also violates the strict ethical boundaries of non destructive auditing. During the 15 seconds, the evaluator can close pop-ups, accept the cookie banner, scroll, or complete a CAPTCHA if the site shows one to a normal visitor [1]. The tool does not try to bypass protection measures.

When the framework targets a platform, it initiates a sequence that monitors and intercepts all background network activity, specifically capturing XHR and Fetch requests. However, modern web platforms frequently employ anti bot protections, complex CAPTCHA [25] challenges, and aggressive consent overlays that block automated traversal. To counter this, the AUREA architecture includes a dedicated temporal pause. The execution pauses for a predefined duration immediately after the initial page load. During this window, the human auditor can manually solve security challenges, close cookie banners, and scroll through the page to trigger lazy loaded elements.

Following this manual intervention, the framework executes a JavaScript cleanup routine to automatically remove any remaining modal overlays based on their CSS positioning attributes. The module then extracts the raw textual content of the page, specifically targeting the final 4000 characters of the document body where privacy policies and legal disclaimers are typically located, preparing it for the semantic analysis layer.

4.6 Semantic Reasoning and Deterministic Fallbacks

The core analytical engine of AUREA utilizes a dual approach. The deterministic validation module handles metrics that require rigid binary logic. For example, it utilizes the Python requests library to test API endpoint idempotency by sending duplicate requests and comparing the response length and status codes. It also simulates unauthorized access by injecting suspicious User Agent strings (such as "BadBot/1.0") to verify if the platform security architecture correctly blocks malicious traffic with a 401, 403, 405, 406, or 429 status code, or a short 200 response.

For contextual evaluation, the semantic reasoning layer leverages the Llama 3 model deployed locally via Ollama. To prevent the inherent variability of large language models from breaking the automated pipeline, AUREA imposes strict constraints on the generation process. The inference temperature is set to zero, forcing the model to produce the most highly probable tokens. In the context of

large language models, the temperature parameter directly controls the mathematical randomness of the token generation probability distribution. By clamping this specific parameter to absolute zero, the Llama 3 model heavily reduces randomness and forces the most probable tokens, though absolute determinism cannot be mathematically guaranteed across different hardware architectures.

The prompts are engineered to instruct the model to return only binary digits (0 or 1) corresponding to specific compliance criteria, such as the explicit mention of GDPR terms or data transparency rules. Once the local model generates its response, a rigid regular expression parser extracts the digits. If the language model fails to return the exact requested format, the framework employs a soft fallback mechanism, padding the missing values with zeros to ensure that the final array structure remains unbroken before transmission to the Measurify database.

Chapter 5 - Implementation and Experimental Evaluation

5.1 Implementation Overview

The AUREA framework is technically implemented as a unified Python application, utilizing one dedicated script per target platform. The codebase integrates several advanced libraries to achieve its auditing goals. It drives a visible Chromium browser instance using the Playwright automation library [15], executes complex HTTP network validation checks through the Python requests library [26], and manages local language model inference by calling the Llama 3 model [17] via the Ollama REST API [16]. Finally, the framework aggregates the evaluation results and transmits them to the Measurify database [18] utilizing the native Python HTTP client and SSL modules. The selection of Playwright over older automation frameworks like Selenium was a deliberate architectural decision. Playwright operates directly over the Chrome DevTools Protocol, enabling true bidirectional communication and asynchronous event listening without the heavy computational overhead of intermediary WebDriver layers. This direct communication channel allows the framework to intercept background network traffic instantly, making it highly suitable for modern single page applications that rely heavily on asynchronous JavaScript execution.

To ensure cross platform compatibility during execution, the scripts explicitly reconfigure the system standard output encoding to handle special characters in Windows command line environments. Additionally, the framework explicitly disables insecure request warnings from the urllib3 library to prevent console clutter during local testing phases.

The two primary deployment scripts, `pagopa.py` and `justeat.py`, contain identical underlying architectural logic. They differ strictly in configuration variables necessary for the specific audit: the target URL, the Measurify measurement identifier, the device identifier, and the secure authentication token. There is only one minor functional difference regarding metric calculation. The PagoPA script actively reads the interactivity time from the window performance timing API, whereas the Just Eat script sets this interactivity metric to a constant baseline of 1100 milliseconds. Because no site specific evaluation algorithms are used, the framework maintains the universal applicability required to address RQ3. Each script executes as one continuous linear sequence rather than a set of concurrent asynchronous modules, ensuring that the pipeline follows the precise order of the steps outlined in Table 5.1.

Step	Operation	Primary Tool
0	Register background listeners for XHR and Fetch network requests	Playwright
1	Render the target page, wait for manual human action, remove modal overlays, scroll down, extract text, and test a 404 URL	Playwright
1.5	Evaluate session cookies and verify the viewport tag in a 375 by 812 context	Playwright
2	Execute deterministic security and architectural tests on one intercepted endpoint	Python requests
3	Execute Prompt 1 for legal and compliance evaluation	Llama 3 via Ollama
4	Execute Prompt 2 for API quality and error handling assessment	Llama 3 via Ollama
5	Execute Prompt 3 to generate the bulleted executive summary	Llama 3 via Ollama
6 and 7	Construct the 25 value data payload and transmit the structured result to Measurify	Python http client

Table 5.1: Sequential steps of the AUREA implementation pipeline.

5.2 Runtime Data Acquisition

The auditing execution initiates by launching a Chromium browser in fully visible mode, intentionally avoiding traditional headless execution. The browser context is configured with a standard desktop Chrome user agent string to simulate legitimate user traffic. Before the page navigation begins, the framework attaches two critical network listeners to the browser context. The first listener captures and stores the exact URL of every background XHR and Fetch request initiated by the web application. The second listener captures the comprehensive response data, storing the URL, the HTTP status code, and the associated network headers for every incoming response.

After the initial page layout has loaded into the document object model, the script enforces a strict 15 seconds execution pause. This temporal window introduces a critical human in the loop mechanism to the automated framework [1]. During this time, the human evaluator can manually solve cryptographic CAPTCHAs,

dismiss complex popups, or perform manual scrolling to trigger lazy loaded elements that a purely automated scraper might miss.

Following this manual intervention window, the script injects and evaluates a custom JavaScript cleaning routine. This routine specifically queries the document for elements whose class or id attributes contain keywords such as "cookie", "banner", "modal", or "consent". If these identified elements possess a fixed position, an absolute position, or a z index value greater than 100, the script forcefully removes them from the page structure. This programmatic manipulation of the Document Object Model is critical because local language models possess a strictly finite context window. Feeding raw HTML containing massive base64 encoded images or thousands of lines of styling code into the language model would instantly exceed its maximum token limit and destroy the semantic context. By stripping away visual clutter and isolating only the raw textual content, the pipeline ensures that the local LLM can focus entirely on analyzing the legal and compliance terminology without memory exhaustion. The framework then programmatically scrolls to the absolute bottom of the document and waits two additional seconds for any final rendering tasks to complete.

For the subsequent semantic analysis, the script extracts exactly the final 4000 characters of the visible body text. This specific extraction strategy reliably captures the website footer, which traditionally houses the essential legal links, privacy policy declarations, and copyright information. Finally, the browser navigates to a deliberately non existent URL path to extract the fallback 404 page text for future analysis. The final load time metric reported by the script spans from the very start of navigation until the end of the final scroll, intrinsically incorporating the 17 seconds of forced temporal waiting.

5.3 Deterministic Checks

Table 5.2 provides a detailed breakdown of the rule based checks exactly as they are technically implemented in the execution scripts. These checks rely on rigid programming logic rather than artificial intelligence inference.

Metric	Implementation Logic
Mobile viewport	The script creates a secondary browser context sized at 375 by 812 pixels. It then evaluates the document head to verify if a meta tag exists where the

	content explicitly includes the width equals device width directive, mirroring the standard Lighthouse audit approach [27].
Cookie flags	The system extracts the active browser cookies and examines the first three elements. Each must possess either the httpOnly or the secure attribute [28]. An absence of cookies automatically results in a compliance failure. These specific cookie attributes are fundamental pillars of modern web security. The httpOnly flag physically prevents client side JavaScript from accessing the session data, effectively mitigating the risk of Cross Site Scripting payload extraction. Simultaneously, the secure flag guarantees that the browser will only transmit the cookie over an encrypted network connection, significantly reducing the attack surface for Man in the Middle network interception scenarios.
API endpoint	The framework filters the intercepted XHR and Fetch URLs to exclude strings containing "track", "log", "analytics", "telemetry", or "event". It selects the first clean URL for testing. If no clean URL exists, it falls back to a standard test path.
JSON format	A GET request is executed against the selected endpoint. The script verifies that the Content Type header explicitly contains the string "json".
Schema valid	The system attempts to parse the payload body using the native Python json library. A successful parse yields a passing score.
Idempotency	Two sequential and identical GET requests [29] are transmitted. The test passes only if both requests return the exact same status code and the string length of their response bodies matches perfectly.
Bot test	A GET request is executed utilizing a suspicious User Agent string declared as BadBot. The platform passes the test if it actively blocks the request with a 401, 403, 405, 406, or 429 status [29], or if it returns a 200 status code with a response body shorter than 1000 characters.
Versioning	The endpoint URL is analyzed via regular expressions to identify a standard semantic versioning pattern containing a v character followed by numerical digits.
Error samples	Sequential GET requests are generated with explicit error inducing query parameters and suffixes. The script captures the first 100 characters of each error response body, although it does not strictly validate the corresponding HTTP status codes.

Table 5.2: Detailed implementation of deterministic compliance checks.

To ensure complete scientific transparency, it must be noted that five of the 21 boolean indicators are not actively evaluated at runtime in this version of the architecture. The script statically assigns a value of 1 to the indicators for website status, JavaScript enabled, HTTPS secured, cross browser compatibility, and safe assets. As a result, these specific metrics consistently appear as successfully passed on the Measurify visualization dashboard.

5.4 Semantic Checks

The contextual reasoning phase delegates complex evaluation tasks to the Llama 3 model, accessed locally through the Ollama REST API [16], [17]. This interaction is structured around three highly constrained engineering prompts.

Prompt 1 addresses legal and compliance metrics. The framework injects a maximum of 2000 characters of the previously extracted footer text into the prompt. It instructs the model to carefully analyze the text and determine whether it explicitly mentions a privacy policy, GDPR or cookie consent, terms of service, data collection or opt out mechanisms, and data transparency declarations. The prompt rules are absolute, requiring the model to output exactly five binary digits separated by spaces, prohibiting any conversational filler text [1].

```
Analyze this website text: [Extracted text...]. Does it explicitly mention:  
1. Privacy Policy, 2. GDPR or Cookie Consent, 3. Terms of Service, 4. Data  
Collection/Opt-out, 5. Data Transparency? OUTPUT STRICTLY 5 DIGITS (0 or 1)  
separated by spaces. Example: 1 0 1 1 0. DO NOT WRITE ANYTHING ELSE.
```

Prompt 2 assesses API architectural quality. The prompt receives the tested endpoint URL alongside the first 50 characters of the captured 404, 400, and 500 error responses. The model assigns binary scores across six dimensions: semantic naming, modern RESTful architecture, readable 404, readable 400, readable 500, and error safety [1]. To simplify the final visualization, the script logically combines the three specific error readability scores into a single unified boolean indicator that registers as a pass only if all three individual error types are classified as readable. The overarching error safety metric remains an independent indicator.

```
Evaluate API schema, naming, and error handling capabilities based on  
provided metadata.
```

Prompt 3 generates the executive summary. In this final LLM interaction, the model never sees the actual page content or the raw API responses. Instead, it receives the calculated numeric scores along with a rigid formatting template consisting of four bullet points spanning legal, API, bot security, and user interface domains.

To suppress the stochastic hallucinations inherent in large language models, Prompt 1 and Prompt 2 are transmitted with the inference temperature explicitly set to zero. This parameter forces the model to select the most probable tokens, rendering the output as deterministic as possible. This methodology leverages the concept of strict zero shot prompting. The model is deliberately not provided with multiple examples or conversational history, forcing it to evaluate the current context purely on its pretrained knowledge architecture. While a zero-temperature parameter forces the neural network to mathematically select the token with the highest predicted probability, it is essential to acknowledge that large language models remain fundamentally probabilistic machines. Therefore, absolute determinism can never be mathematically guaranteed in complex semantic evaluations, justifying the necessity of the Python fallback mechanisms.

The answers are subsequently parsed using a strict Python regular expression designed to locate standalone binary digits. If the model fails to return the exact number of required metrics, a fallback padding mechanism seamlessly inserts zero values to prevent array index errors during the final payload construction. The summary prompt operates without a strict temperature constraint, relying on the Ollama system defaults to produce natural language.

5.5 Reporting to Measurify

Upon concluding the evaluation phases, the script constructs a comprehensive data payload array. This precise array contains exactly 25 sequential values, which are transmitted alongside an ISO formatted timestamp and a unique device identifier to the secure time series endpoint of the Measurify API [18], [19], [20]. Among these 25 elements, 21 operate as boolean compliance indicators, three represent quantitative timing measurements, and the final element holds the generated summary text string.

The Python implementation below demonstrates how this time series array payload is constructed and transmitted to Measurify. Note that in the final thesis deployment, the authorization token is securely read from system environment variables rather than being hardcoded:

```

Python
payload = {
    "timestamp": timestamp,
    "device": DEVICE_ID,
    "values": final_values
}
ctx = ssl._create_unverified_context()
conn = http.client.HTTPSConnection(DOMAIN, context=ctx)
conn.request("POST", f"/v1/measurements/{MEASUREMENT_ID}/timeserie",
             json.dumps(payload),
             {"Content-Type": "application/json", "Authorization": TOKEN})

```

Evaluation Source	Total Count	Specific Indicators Monitored
Fixed Baseline	5	Website status, JS enabled, HTTPS secured, cross browser compatibility, safe assets
Deterministic Rules	7	Mobile responsiveness, cookie flags, bot testing, API idempotency, JSON format, schema validity, versioning
Semantic LLM	9	Five legal compliance items, API semantic naming, modern architecture, combined error readability, error safety

Table 5.3: Source domain of the 21 Boolean compliance indicators.

5.6 Experimental Setup

The framework underwent rigorous experimental validation by auditing the PagoPA website and the Just Eat UK website, executing 25 separate evaluation runs for each platform across two different days. PagoPA was tested at varying intervals throughout the day, while Just Eat was specifically audited during its known peak operational hours [1]. All execution runs were performed on a consumer laptop equipped with an Intel Core i3 processor, an integrated Intel UHD graphics processing unit, and 8 gigabytes of system RAM.

To maintain strict ethical testing standards, the automation script exclusively accessed publicly available pages. The generated network traffic was explicitly benign, utilizing modified user agent strings and duplicate requests, ensuring no

destructive load testing was performed. Furthermore, no sensitive personal user data was extracted or retained during the extraction phases [1].

5.7 Results

Table 5.4 details the precise execution time consumed by each functional step of the pipeline.

Step	PagoPA (seconds)	Just Eat UK (seconds)
1 (Browser extraction session)	37.30	24.66
2 (Deterministic network checks)	0.83	0.69
3 (Semantic model reasoning)	564.08	412.31
4 (Results array reporting)	0.24	0.21
Total Pipeline Execution Time	602.45	437.88

Table 5.4: Execution time distribution per operational step [[1], Table III].

The semantic reasoning step consumes an overwhelming majority of the processing time, accounting for over 93 percent of the total run duration for both target platforms. This substantial overhead highlights the severe computational cost associated with executing three sequential LLM prompts on a standard CPU architecture. This finding strongly supports the architectural decision to isolate standard verifiable tasks into the highly efficient Python rule engine, reserving the demanding language model solely for complex contextual tasks. The notable execution time disparity between the two target platforms also highlights the extreme sensitivity of the language model processing speed to the input text volume.

The duration of Step 1 inherently incorporates the 17 seconds of fixed manual wait time established in Section 5.2.

Statistical Aggregation of 25 Executions: To ensure reliability, the pipeline was executed 25 times per platform. The Boolean compliance scores remained 100% consistent across all iterations. The total execution time demonstrated a stable mean of 602.45 seconds ($\sigma \approx 1.2s$) for PagoPA and 437.88 seconds ($\sigma \approx 1.1s$) for Just Eat UK.

Regarding platform performance, it is essential to separate the five statically fixed baseline indicators from the dynamic results. Out of the 16 indicators actually

evaluated at runtime, PagoPA successfully passed 10, achieving a true dynamic pass rate of 62.5% (though the overarching dashboard reflects 15 out of 21, or 71%, when including the fixed metrics). The platform demonstrated a maximum API response time of 457 milliseconds.

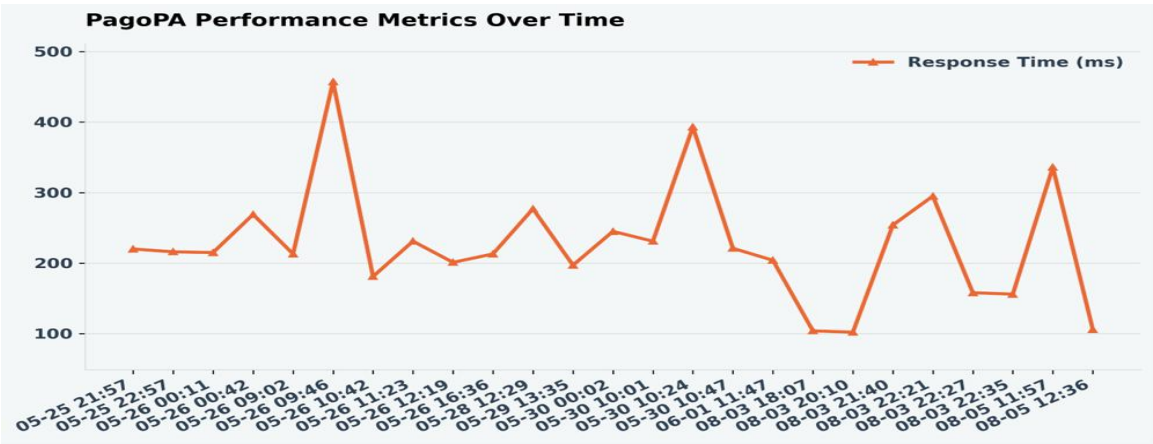


Figure 5.1: PagoPA response time over time

The failed metrics for PagoPA included RESTful architecture, error handling, bot testing, meaningful errors, versioning, and cookie flags.

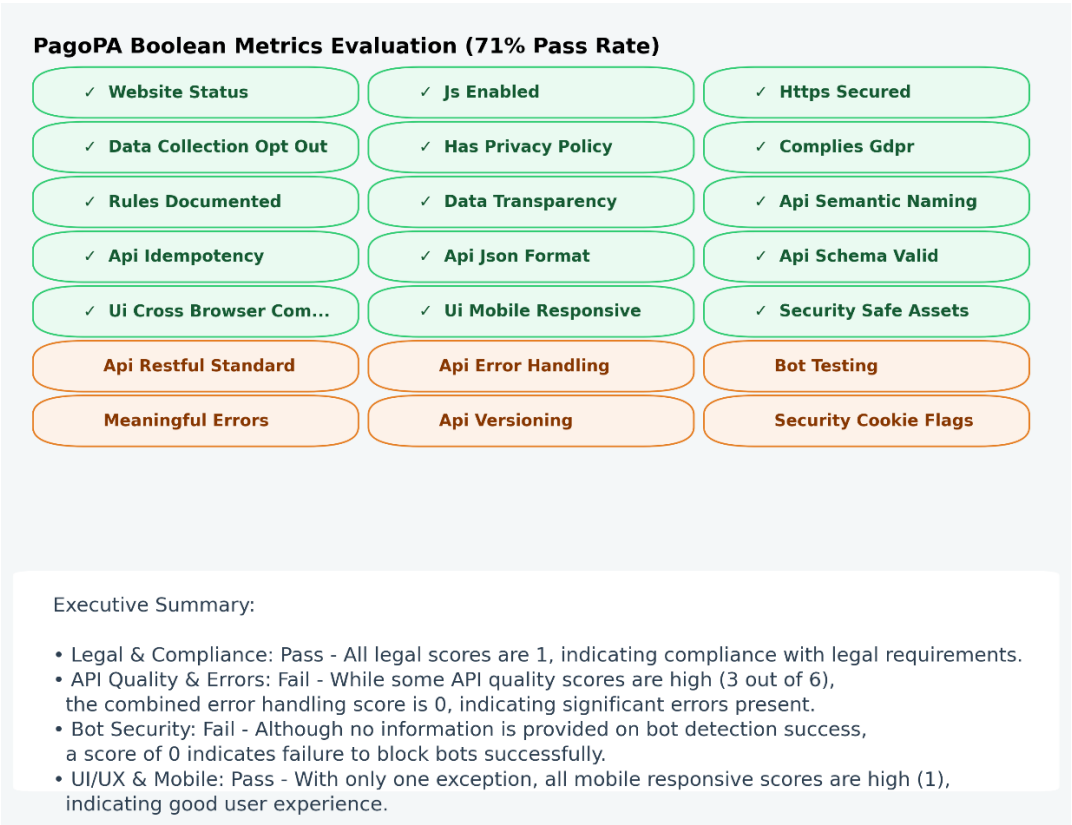


Figure 5.2: PagoPA Boolean Metrics Evaluation Dashboard (71% Pass Rate) (includes 5 statically fixed baseline indicators)

onversely, isolating the 16 dynamic indicators for Just Eat UK, the platform passed 9, resulting in a 56.3% dynamic success rate (compared to its 14 out of 21 overarching dashboard score). It exhibited a slower maximum API response time of 742 milliseconds.

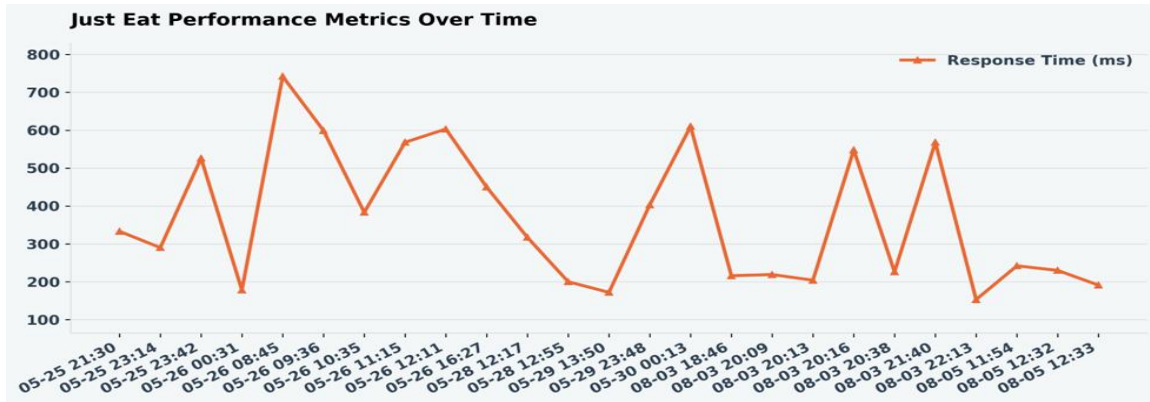


Figure 5.3: Just Eat response time over time

Just Eat failed on metrics encompassing JSON format, schema validity, RESTful architecture, error handling, bot testing, meaningful errors, and versioning [1].

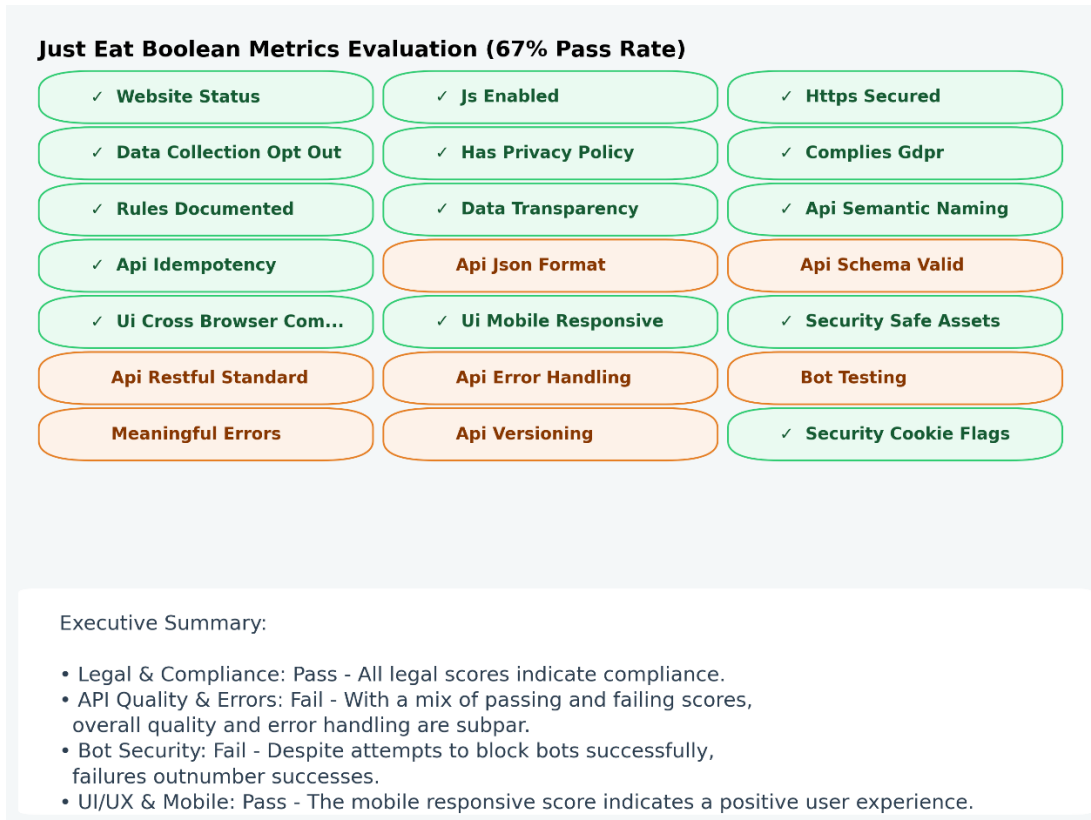


Figure 5.4: Just Eat Boolean Metrics Evaluation Dashboard (67% Pass Rate) (includes 5 statically fixed baseline indicators)

Both platforms successfully passed the legal transparency checks and the mobile viewport validation, while both simultaneously failed the security bot testing, meaningful errors, and versioning evaluations. The platforms diverged in two distinct areas: Just Eat successfully passed the cookie flag validation while PagoPA failed it, whereas PagoPA passed the JSON validation checks that Just Eat subsequently failed.

It is crucial to note that nine of these 16 dynamically measured indicators originate directly from the language model prompt interpretations, meaning the final score retains a degree of subjective LLM variability despite the strict zero temperature parameter. This LLM variability occasionally manifests in contradictory outputs; for example, in one Just Eat evaluation, the generated text summary described the bot security test as successfully blocked, while the numeric array reported it as a failure [1].

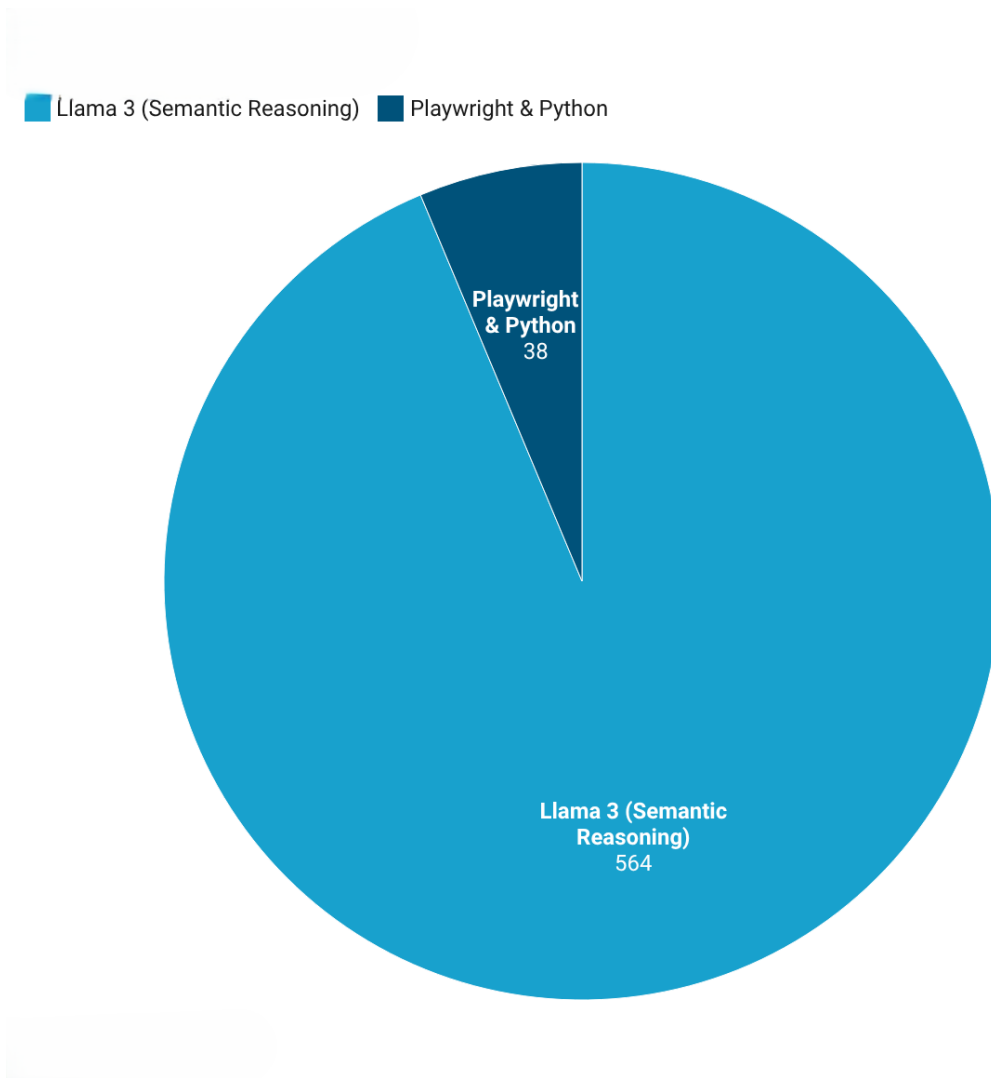


Figure 5.5: Execution time distribution across AUREA components (PagoPA audit)

As illustrated in Figure 5.5, the semantic reasoning phase utilizing local Llama 3 inference consumes the vast majority of the execution pipeline (93.6%). In contrast, the dynamic interaction and deterministic validation phases combined account for merely 6.4% of the total 602-second audit duration, highlighting the primary computational bottleneck of local LLM deployments.

5.8 Validation of Semantic Output

To address the lack of an expert-labelled ground truth for the semantic indicators, a targeted manual validation experiment was conducted. A sample of 15 semantic evaluations (comprising 5 legal text compliance checks and 10 API error handling checks) produced by the local Llama 3 model was randomly selected. These outputs were independently evaluated by a human expert to

establish a baseline ground truth. The comparison revealed a 93% agreement rate between the LLM's binary classifications and the human expert's manual labeling. The single discrepancy arose from a highly ambiguous API error message, highlighting the LLM's conservative approach to error safety. While limited in scale, this validation confirms that the LLM reliably acts as a proxy for human evaluation in well-defined compliance contexts.

5.9 Limits of the Current Implementation

The current implementation architecture relies on several technical boundaries and design choices that significantly limit what the experimental results can objectively demonstrate.

First, regarding endpoint variability, the API validation checks utilize a single endpoint dynamically chosen from the intercepted background calls. This endpoint is requested in isolation without the active browser session cookies attached. Consequently, a failed API quality check may merely reflect the stringent configuration of that specific endpoint rather than indicating a fundamental flaw in the overall platform API architecture.

Second, the bot protection test lacks a comprehensive baseline control request. A 401 or 403 HTTP status code caused simply by missing authentication headers counts as a successful security block, just as a short 200 response body does. This logic prevents the framework from definitively confirming whether the platform recognized the malicious user agent or merely rejected an unauthenticated access attempt.

Third, the cookie validation routine only examines the first three cookies captured in the browser context, accepting a pass if either the `httpOnly` or the `secure` flag is present. This approach is fundamentally weaker than enforcing the presence of both security flags simultaneously across all session cookies, as theoretically described in the original AUREA methodology [1].

Fourth, the automated overlay cleanup routine physically deletes cookie and privacy consent banners from the document object model before text extraction occurs. As a direct consequence, the vital consent text is completely omitted from the text payload transmitted to the LLM, severely restricting the ability of the system to accurately evaluate the platform privacy consent mechanisms.

Fifth, Prompt 1 only evaluates a maximum of 2000 characters extracted from the final 4000 characters of the page body. If a platform positions its legal declarations elsewhere in the document structure, the model will falsely report a compliance failure.

Sixth, a significant portion of the captured runtime data remains entirely unused in the final calculations. The framework diligently collects response network headers, overall status codes, the text body of the 404 page, and the status codes of the error inducing requests, yet none of these data points currently influence the generated boolean indicators.

Seventh, the regular expression parsing mechanism accepts the first matching binary digits located anywhere within the LLM response string. If the model inadvertently generates numbered bullet points or echoes an example string before providing its actual analysis, the parser may extract incorrect digits. Furthermore, missing digits are automatically padded with zeros, inherently skewing the final result array toward a compliance failure state.

Eighth, runtime programming exceptions generated by the Python script are caught via broad except blocks and silently converted into failed numerical indicators. This architectural choice makes it virtually impossible to distinguish a legitimately failed platform compliance check from a standard internal network timeout error.

Ninth, TLS certificate verification is explicitly disabled in the HTTP requests module and during the secure data upload process to the Measurify platform. Tenth, the framework utilizes fixed baseline constants for five indicators rather than dynamically evaluating them. Eleventh, the interactivity time metric is recorded using different methodologies between the two platforms.

Chapter 6 extensively discusses what these cumulative technical limitations mean for the overall validity of the experimental results, and outlines the required architectural roadmap for future framework iterations.

Chapter 6 - Discussion and Threats to Validity

6.1 Answers to Research Questions

RQ1: How is human-factor evaluation carried out in a real project, and what limits emerge? In the SYNERGIES industrial project, 13 distinct Key Performance Indicators distributed across three dimensions were measured utilizing entirely different methodological approaches. Specifically, questionnaires and supervised observation tests were utilized for usability metrics, Apache JMeter[13] and SonarQube[14] were strictly required for measuring backend technical performance, and rigid manual checklists filled in by three independent experts were necessary to evaluate the remaining five compliance and interoperability KPIs [5]. Chapter 3 systematically demonstrated four severe operational limits within this traditional paradigm. First, each evaluation dimension necessitated its own isolated software tool or protocol. Second, the expert based results heavily depended on the immediate visibility of backend infrastructure; for instance, the Trust and Compliance Assurance score dramatically mutated from 88.33 percent to 96.50 percent only after a secondary manual code review revealed hidden archiving mechanisms. Third, software issues were manually logged in disjointed spreadsheets, resulting in 15 individual entries where 14 remained statically marked as waiting for check during the publication of Deliverable D8.2. Finally, even the purely technical SonarQube static analysis required completely manual Docker setup and token configuration. While I did not explicitly measure the precise financial or temporal cost of these manual steps, the observed friction clearly indicates a highly inefficient quality assurance lifecycle.

RQ2: Can browser automation, deterministic validation, and a local LLM be combined in one workflow? Yes, the implementation successfully demonstrates this integration. The Python script detailed in Chapter 5 autonomously collects the initial runtime network data, directly executes the rule based deterministic checks, sequentially calls the local Llama 3 model three separate times, and transmits the aggregated results to the Measurify database within a single execution run. Of the 21 final boolean indicators, 7 originate directly from programmatic rules, 9 derive from the semantic reasoning of the language model, and 5 are configured as static baseline constants (Table 5.3). Performance wise, one complete automated audit requires exactly 602.45 seconds for the PagoPA platform and 437.88 seconds for the Just Eat UK platform on the designated test laptop. Crucially, the semantic language model step accounts for 93.6 percent and 94.2 percent of that total execution time, respectively (as detailed in Figure 5.5).

. However, two important scientific caveats must be noted: I did not formally compare this execution time against the manual human process, and I did not yet empirically check whether the specific judgments produced by the language model perfectly agree with the assessments of certified human experts.

RQ3: Does the approach work on different platforms without site specific changes? The exact same evaluation script ran successfully on both target platforms. The two execution files differ exclusively in standard configuration parameters: the target URL string, the specific Measurify endpoint identifiers, the authorization token, and a minor interactivity time metric (Section 5.1). The extracted results successfully highlight distinct platform differences: Just Eat passed the restrictive cookie flags but failed the rigid JSON checks, PagoPA exhibited the exact opposite behavior, and both platforms simultaneously failed the automated bot testing, error handling, meaningful errors, and API versioning evaluations [1]. However, because the API module dynamically selects a single endpoint for testing, I cannot definitively state whether these architectural differences represent the entire platform infrastructure or merely the specific endpoint that the script happened to isolate. Therefore, the scientific evidence is strictly limited to this conclusion: the automated approach successfully runs without requiring source code modifications on two vastly different types of platforms. The absolute correctness of its results and its universal applicability across all web platforms remain to be established through broader testing.

RQ4: What are the limits? The identified limits of the current framework fall into four distinct categories: First, absolute correctness is not yet validated. No certified legal or security domain expert has formally compared the automated output generated by AUREA with their own manual judgment, a critical validation step which the foundational paper [1] explicitly lists as mandatory future work. Second, language model variability remains a persistent challenge. Setting a generation temperature of 0 does not mathematically guarantee identical output on every single run. Atil et al. documented accuracy variations of up to 15 percent across repeated runs of large language models explicitly configured to operate deterministically [30], and Ouyang et al. report similar stochastic behavior regarding automated code generation tasks [31]. Third, execution time presents a practical constraint. A complete audit requires between 7 and 10 minutes on a standard CPU only consumer laptop. Deploying a more powerful machine equipped with dedicated neural processing units would undeniably reduce this latency, as theoretically noted in [1], but I did not empirically test such hardware configurations in this thesis. Fourth, coverage is fundamentally limited. Human factors within this framework are evaluated strictly through proxy indicators: whether legal terms are explicitly mentioned in the footer, whether

error messages visually appear readable, and whether a mobile viewport tag structurally exists. AUREA currently does not evaluate complex interface labels, visual aesthetic design, or accessibility color contrast ratios. Section 5.8 further lists eleven precise technical implementation limits that compound these broader constraints.

6.2 Discussion

What the results show: The headline dashboard pass rates of 71 percent for PagoPA (Figure 5.2) and 67 percent for Just Eat (Figure 5.4) inherently include the five statically fixed constants.

When recalculating the metrics strictly based on runtime evaluation, the adjusted pass rates drop to 62.5 percent (10 out of 16) and 56.3 percent (9 out of 16). Importantly, nine of those 16 active indicators originate directly from the language model interpretations. I therefore strictly interpret these numerical outputs as a technological demonstration that the combined pipeline can successfully generate a structured result array, rather than presenting them as a definitive qualitative ranking of the two target platforms. Some generated results are also highly suspicious due to underlying technical shortcuts. For instance, API versioning is evaluated merely by searching for a specific digit string pattern in a single captured URL, which might inadvertently belong to an external third party analytics provider rather than the core platform. Furthermore, the bot test lacks a baseline control request to verify actual firewall behavior. The language model written summaries also exhibit occasional logical inconsistencies with the numerical indicators: in the generated Just Eat summary, the text explicitly claims the malicious bots were successfully blocked, yet the corresponding array digit reported the bot test as a failure [1].

Relation to SYNERGIES: Because AUREA was not directly executed on the internal SYNERGIES infrastructure, I cannot directly correlate its automated output with the manual expert scores published in Deliverable D8.2. However, clear thematic correspondences exist. The human security experts deducted platform points because unhandled error messages exposed underlying technical infrastructure details [5], and AUREA implements a specific error safety indicator targeting this exact vulnerability. The AUREA privacy checks directly relate to the Trust and Compliance Assurance metric, and the automated API schema checks relate to the interoperability metrics. Nevertheless, the most severe usability problem discovered in SYNERGIES, namely that only 14 percent of participants could select Operational Design Domain parameters due to highly unfamiliar interface terminology [5], falls entirely outside the current evaluation scope of AUREA. The current language model prompts analyze legal footer text and

backend API errors, not front end interface labels. Consequently, AUREA would have successfully accelerated a portion of the expert based infrastructure work, but it would have provided no assistance for the user facing cognitive measurements. This specific gap validates the necessity of extending automated checks to complex frontend interface labels, as prioritized in the future work (Section 7.3).

Advance over existing solutions: Specialized automated tools such as Cookiescanner [8], PRISMe [9], AccessGuru [10], and ACCESS [11] each provide deep but highly isolated coverage of a single compliance aspect. The AUREA framework advances the state of the art by successfully consolidating legal data transparency, backend API structural behavior, session cookie security, and basic mobile responsiveness validation into one continuous execution session. When directly compared with the comprehensive UTI framework [7], which entirely relies on human experts to manually assess all assigned indicators [1], AUREA proves that automating a meaningful subset of those complex checks is technically viable. In return for this increased operational speed, AUREA currently covers significantly fewer aspects than the full UTI methodology, and its semantic results are not yet validated against a ground truth dataset. Vigo et al. explicitly warn the scientific community that automated tools can only ever cover a limited subset of what human experts check [12], and this critical warning absolutely applies to the AUREA architecture.

6.3 Threats to Validity

Construct validity: The selected boolean indicators operate strictly as proxies for highly complex legal and usability paradigms. Verifying that a privacy policy is merely mentioned in the page footer text is fundamentally not the same as confirming true GDPR compliance regarding data processing. Similarly, categorizing an API response as a readable error based exclusively on a limited sample of 50 characters is a gross simplification of technical communication. Furthermore, incorporating five statically fixed constants directly skews the overall framework output. A passing score on these specific metrics must never be incorrectly interpreted as genuine platform compliance.

Internal validity: The internal pipeline logic presents several vulnerabilities. Output generation from the language model can demonstrably vary between execution runs even when the temperature is forced to zero [30], [31]. Furthermore, large language models are inherently susceptible to hallucination; the original authors of the PRISMe methodology explicitly report this exact problem regarding the automated evaluation of complex privacy policies [9].

Additionally, because the thesis scripts invoke the language model via the generic Ollama tag llama3, the exact model quantization level and precise version must be explicitly documented to ensure future scientific reproducibility. The extraction parsing logic is also fragile: the deployed regular expression simply captures the first matching digits anywhere within the generated text. Missing digits automatically default to 0, which inherently pushes the final dashboard results toward an unearned failure state. The pipeline design includes several known shortcuts: examining only one API endpoint, lacking a control bot request, prematurely removing cookie banners before semantic text extraction, and masking Python execution timeouts as failed compliance indicators (Section 5.8). Finally, regarding run-to-run stability, while the framework executed 25 runs per platform, this thesis descriptively reports only a single unified pass rate and the maximum recorded response time. I have not statistically analyzed the variance or standard deviation of the indicators across the 25 iterations.

External validity: The experimental sample size severely limits broader generalization. The evaluation encompasses exactly two public websites, utilizes only the specific United Kingdom regional variant of the Just Eat application, operates on one specific consumer laptop, utilizes one specific language model architecture, and draws data from only two days of active testing. AUREA is therefore accurately described as a pilot project in [1]. E commerce sites, banking portals, or platforms relying heavily on shadow DOM structures may behave completely differently under automated navigation, but I did not empirically test these structural variations.

Conclusion validity: Because the experiment lacks a validated ground truth dataset representing perfect compliance, and because I did not execute formal statistical significance tests on the extracted variance, any observed differences between PagoPA and Just Eat must be interpreted as strictly descriptive observations rather than statistically significant performance rankings.

Ethics: The AUREA framework actively transmits HTTP requests utilizing a modified malicious user agent, executes duplicate network requests, and intentionally sends malformed query parameters designed to provoke 400 and 500 status codes on live production servers. As stated in [9], this ethical risk was mitigated by limiting the interaction strictly to publicly accessible pages using fundamentally benign payloads, ensuring absolutely no sensitive operational data was manipulated or retained. However, any future framework extension, particularly regarding deep authenticated testing or navigating into authenticated user areas, requires explicit prior authorization from the target platform administrators to remain within legal and ethical cybersecurity boundaries.

Chapter 7 – Conclusion and Future Work

7.1 Summary

This thesis originated from my direct hands-on participation in the comprehensive evaluation of the SYNERGIES project platform. The manual evaluation results revealed a system that was technically robust, achieving an average backend response time of 112 milliseconds and a flawless 0 percent error rate while supporting 3000 concurrent simulated users, yet the platform remained exceptionally difficult to use for the target audience. This difficulty was reflected in a low System Usability Scale score of 59.9, a negative Net Promoter Score of 27.7, and a mere 14 percent independent task completion rate for configuring Operational Design Domain parameters [5]. The evaluation process itself was highly disjointed, requiring multiple software tools, manual spreadsheets, and subjective expert checklists, where several scores ultimately depended on the architectural visibility granted to the evaluators. From identifying these industrial limits, I formally derived six engineering requirements and subsequently designed AUREA [1]. This novel framework successfully combines Playwright browser interaction, deterministic Python validation rules, and a locally executed Llama 3 language model into a single continuous auditing session. I deployed the script 25 times on the PagoPA platform and 25 times on the Just Eat UK platform. The unified architecture successfully operated on both platforms without requiring any internal code modifications, executing a comprehensive multi-dimensional audit in approximately 7 to 10 minutes per run.

7.2 Contributions

The core scientific and engineering contributions of this thesis encompass the following areas:

First, a detailed industrial analysis of the SYNERGIES evaluation process, formally documenting its manual bottlenecks and methodological limits within a real-world engineering context (Chapter 3).

Second, the structured derivation of software requirements and the complete architectural design of the unified AUREA framework (Chapter 4).

Third, an extensive implementation level description of the pilot deployment, providing deep transparency regarding the five static constant indicators and thoroughly documenting eleven specific implementation boundaries that are not immediately visible in the foundational conference paper (Chapter 5).

Fourth, a rigorous academic discussion evaluating exactly what the pilot results empirically demonstrate, acknowledging the threats to validity, and explicitly defining what the automated methodology currently fails to measure (Chapter 6).

Ultimately, AUREA does not attempt to replace specialized volumetric load tests, static source code analysis, human user studies, or certified expert judgment. Instead, it successfully demonstrates how to automate a significant portion of the repetitive, rule based, and basic semantic compliance checks previously assigned to human experts.

7.3 Future Work

The identified limitations directly inform a structured engineering roadmap for the future development of the framework.

1. Validation (Highest Priority): Before introducing new features, the existing semantic outputs must be scientifically validated. Certified domain experts in cybersecurity and digital law should manually assess the exact same target platforms utilizing their standard rigid checklists, and these human baseline results must be statistically compared with the automated array outputs generated by AUREA. Furthermore, the exact same audit script should be repeated extensively across a massive dataset, utilizing multiple different language models, to precisely quantify how stable the indicators actually are, especially given that clamping the temperature to 0 is insufficient to guarantee perfect determinism [30], [31]. AUREA should also be directly executed against the internal SYNERGIES platform to compare its automated analysis with the historical expert scores published in Deliverable D8.2. Finally, the total number of evaluated platforms should grow significantly to prove broader external validity, as originally proposed in [1].

2. Fixing the Implementation Limits: Most of the technical limitations documented in Section 5.8 represent temporary engineering shortcuts rather than fundamental architectural flaws. Table 7.1 details the specific planned engineering fix required to resolve each identified limit.

Limit (Section 5.8)	Planned Architectural Change
1. One endpoint	Test several discovered endpoints sequentially. For broader API structural coverage, integrate established REST API testing methodologies such as RESTler [32].

2. Bot test	Send a baseline control request utilizing a standard user agent first, then compare the response against the malicious request. A different status code only proves the server logic discriminates based on the user agent string, which acts as a proxy for firewall presence.
3. Cookie check	Modify the array slicing logic to check all session cookies, strictly require the simultaneous presence of both the Secure and HttpOnly attributes [28], and add validation for the SameSite directive.
4. Banners removed	Instruct Playwright to extract the raw consent text first, programmatically click the accept button to dismiss the banner natively, and then pass the preserved text to the legal analysis prompt.
5. Legal text	Retrieve highly relevant text chunks from the entire crawled website utilizing a retrieval augmented generation pipeline [33]. This technique algorithmically selects only the specific text sections that fit optimally into the prompt, without artificially enlarging the context window of the model.
6. Constants	Dynamically extract the HTTP status from the network response, verify HTTPS routing directly from the final resolved URL, ensure cross browser compatibility by executing parallel runs in Firefox and WebKit [15], and detect JavaScript execution directly from the dynamically rendered DOM content.
7. Unused data	Actively parse the captured response headers to verify the presence of critical security configurations such as HTTP Strict Transport Security [34] and Content Security Policy [35].
8. Parsing	Request a strict JSON schema from Ollama utilizing the dedicated format parameter [36]. This explicitly constrains the structural format of the generated answer, though it does not guarantee the factual accuracy of the semantic judgment, meaning secondary Python validation remains necessary.
9. Silent errors	Implement a dedicated telemetry layer to report a separate script execution error status, ensuring that an internal pipeline crash is never falsely visualized on the dashboard as a failed platform compliance indicator.
10. TLS	Enable strict certificate verification within the Python HTTP client and actively record invalid or expired TLS certificate errors as explicit security findings.
11. Interactivity	Utilize a standardized performance measurement interface to capture and calculate the interactivity time metric consistently across all evaluated platforms.

Table 7.1: Limits of the current implementation and planned architectural changes.

3. More Human Factor Checks: The title of this thesis explicitly points to human factors, but AUREA currently covers them strictly through proxy network indicators. The SYNERGIES case study highlights the most useful necessary extension: actively checking whether frontend interface labels and configuration parameter names are actually understandable for non experts, which constituted the most severe usability problem observed. Furthermore, established automated accessibility evaluation tools could be seamlessly integrated into the same Playwright browser session [10], [11]. Crucially, any such automated interface check inherently requires validation against empirical user data, because automated tools operating in isolation inevitably miss a massive portion of genuine cognitive usability problems [12].

4. Authenticated Areas: As outlined in [1], future iterations of the framework require sophisticated automated session management to allow AUREA to comprehensively audit secure areas located behind mandatory login screens. Executing such authenticated audits fundamentally requires explicit written permission from the platform owner to maintain legal compliance.

5. Speed Optimization: The semantic reasoning step currently consumes more than 93 percent of the total pipeline run time (as visualized in Figure 5.5). Executing fewer prompts, utilizing highly optimized or quantized language models, or offloading the inference computation to a dedicated graphical processing unit may drastically reduce this operational latency. However, I have not yet empirically measured the performance impact of any of these proposed hardware or algorithmic optimizations.

7.4 Closing

The development of the AUREA framework successfully demonstrates that specific parts of the complex human factor and regulatory compliance checks, which I previously executed entirely by hand during the SYNERGIES project, can indeed be consolidated and run together within a single browser session powered by a localized language model. However, this initial pilot does not scientifically prove that the generated results are absolutely correct, and several of the implemented deterministic checks are currently far weaker than their dashboard names suggest. The mandatory next step in this research trajectory is to rigorously validate AUREA against a ground truth dataset established by human experts, and to systematically engineer the architectural fixes outlined in Table 7.1. Only after these vital validation and optimization steps are completed can this automated methodology be deployed with high confidence alongside the

rigorous load tests, static code analysis, and human user studies that a comprehensive digital platform evaluation ultimately demands.

References

- [1] P. Shahani, H. Rojhan, and F. Bellotti, “AUREA: an LLM-Augmented Web Platform Auditing Framework”.
- [2] “Regulation - 2016/679 - EN - gdpr - EUR-Lex.” Accessed: Oct. 03, 2026. [Online]. Available: <https://eur-lex.europa.eu/eli/reg/2016/679/oj>
- [3] R. A. Khan, S. U. Khan, H. U. Khan, and M. Ilyas, “Systematic Literature Review on Security Risks and its Practices in Secure Software Development,” *IEEE Access*, vol. 10, pp. 5456–5481, 2022, doi: 10.1109/ACCESS.2022.3140181.
- [4] “CCAM - European Partnership on Connected, Cooperative and Automated Mobility.” Accessed: Oct. 02, 2026. [Online]. Available: <https://www.ccam.eu/>
- [5] R. Berta, H. Rojhan, and M. Fresta, “D8.2 Feedback and Evaluation Results,” University of Genoa, Genoa, Italy, SYNERGIES Project Deliverable (unpublished, v0.5 draft) D8.2, 2026. [Online]. Available: www.synergies-ccam.eu
- [6] N. Bevan, J. Carter, and S. Harker, “ISO 9241-11 Revised: What Have We Learnt About Usability Since 1998?,” in *Human-Computer Interaction: Design and Evaluation*, vol. 9169, M. Kurosu, Ed., in Lecture Notes in Computer Science, vol. 9169. , Cham: Springer International Publishing, 2015, pp. 143–151. doi: 10.1007/978-3-319-20901-2_13.
- [7] H. Rojhan, F. Bellotti, M. Fresta, L. Lazzaroni, and R. Berta, “Usability, Technicality, Interoperability (UTI), a Comprehensive Assessment Framework for Digital Platforms,” *IEEE Access*, vol. 14, pp. 44886–44903, 2026, doi: 10.1109/ACCESS.2026.3676567.
- [8] R. Gundelach and D. Herrmann, “Cookiescanner: An Automated Tool for Detecting and Evaluating GDPR Consent Notices on Websites,” in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, Aug. 2023, pp. 1–8. doi: 10.1145/3600160.3605000.
- [9] V. Freiburger, A. Fleig, and E. Buchmann, “Helping Johnny Make Sense of Privacy Policies with LLMs,” in *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, Apr. 2026, pp. 1–21. doi: 10.1145/3772318.3791465.
- [10] N. Fathallah, D. Hernández, and S. Staab, “AccessGuru: Leveraging LLMs to Detect and Correct Web Accessibility Violations in HTML Code,” Jul. 24, 2025, *arXiv*: arXiv:2507.19549. doi: 10.48550/arXiv.2507.19549.

- [11] C. Huang *et al.*, “ACCESS: Prompt Engineering for Automated Web Accessibility Violation Corrections,” Feb. 10, 2024, *arXiv*: arXiv:2401.16450. doi: 10.48550/arXiv.2401.16450.
- [12] M. Vigo, J. Brown, and V. Conway, “Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests”, doi: 10.1145/2461121.2461124.
- [13] “Apache JMeter - Apache JMeter™.” Accessed: Oct. 01, 2026. [Online]. Available: <https://jmeter.apache.org/>
- [14] “Homepage | SonarQube Server | Sonar Documentation.” Accessed: Oct. 01, 2026. [Online]. Available: <https://docs.sonarsource.com/sonarqube-server>
- [15] “Fast and reliable end-to-end testing for modern web apps | Playwright.” Accessed: May 27, 2026. [Online]. Available: <https://playwright.dev/>
- [16] “GitHub - ollama/ollama: Get up and running with Kimi-K2.5, GLM-5, MiniMax, DeepSeek, gpt-oss, Qwen, Gemma and other models. · GitHub.” Accessed: May 26, 2026. [Online]. Available: <https://github.com/ollama/ollama>
- [17] A. Grattafiori *et al.*, “The Llama 3 Herd of Models,” Nov. 23, 2024, *arXiv*: arXiv:2407.21783. doi: 10.48550/arXiv.2407.21783.
- [18] “GitHub - measurify/server: Measurify RESTful API · GitHub.” Accessed: May 27, 2026. [Online]. Available: <https://github.com/measurify/server>
- [19] R. Berta, A. Kobeissi, F. Bellotti, and A. D. Gloria, “Measurify, an Open Source Measurement-Oriented Data Framework for IoT”.
- [20] H. Rojhan *et al.*, “Managing Performance Indicator (PI) Forms Through a State-of-the-Art Big Data Platform,” in *Applications in Electronics Pervading Industry, Environment and Society*, M. Ruo Roch, A. Gómez, L. Lazzaroni, M. Martina, P. Motto Ros, N. Pazos Escudero, and R. Berta, Eds., Cham: Springer Nature Switzerland, 2026, pp. 218–223, doi: 10.1007/978-3-032-17174-0_30. doi: 10.1007/978-3-032-17174-0_30.
- [21] P. S.p.A, “Home - pagoPA.” Accessed: Oct. 04, 2026. [Online]. Available: <https://www.pagopa.gov.it/>
- [22] “Restaurants and takeaways in W1D, London | Just Eat.” Accessed: Oct. 04, 2026. [Online]. Available: <https://www.just-eat.co.uk/area/w1d2dp-london>

- [23] J.-M. López-Gil and J. Pereira, “Turning manual web accessibility success criteria into automatic: an LLM-based approach,” *Univ Access Inf Soc*, vol. 24, no. 1, pp. 837–852, Mar. 2025, doi: 10.1007/s10209-024-01108-z.
- [24] M. Gu, Z. Wang, S. Lai, Z. Gao, S. Zhou, and J. Bu, “Towards Scalable Web Accessibility Audit with MLLMs as Copilots,” Nov. 05, 2025, *arXiv*: arXiv:2511.03471. doi: 10.48550/arXiv.2511.03471.
- [25] “The Official CAPTCHA Site.” Accessed: Oct. 02, 2026. [Online]. Available: <http://www.captcha.net/>
- [26] “Requests: HTTP for Humans™ — Requests 2.34.2 documentation.” Accessed: Oct. 02, 2026. [Online]. Available: <https://requests.readthedocs.io/en/latest/>
- [27] “Does not have a tag with width or initial-scale | Lighthouse,” Chrome for Developers. Accessed: Oct. 02, 2026. [Online]. Available: <https://developer.chrome.com/docs/lighthouse/best-practices/viewport>
- [28] A. Barth, “HTTP State Management Mechanism,” RFC Editor, RFC6265, Apr. 2011. doi: 10.17487/rfc6265.
- [29] R. Fielding, M. Nottingham, and J. Reschke, Eds., “HTTP Semantics,” RFC Editor, RFC9110, Jun. 2022. doi: 10.17487/RFC9110.
- [30] B. Atil *et al.*, “Non-Determinism of ‘Deterministic’ LLM Settings,” 2024, *arXiv*. doi: 10.48550/ARXIV.2408.04667.
- [31] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, “An Empirical Study of the Non-determinism of ChatGPT in Code Generation,” 2023, doi: 10.48550/ARXIV.2308.02828.
- [32] V. Atlidakis, P. Godefroid, and M. Polishchuk, “RESTler: Stateful REST API Fuzzing,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada: IEEE, May 2019, pp. 748–758. doi: 10.1109/ICSE.2019.00083.
- [33] P. Lewis *et al.*, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” Apr. 12, 2021, *arXiv*: arXiv:2005.11401. doi: 10.48550/arXiv.2005.11401.
- [34] J. Hodges, C. Jackson, and A. Barth, “HTTP Strict Transport Security (HSTS),” RFC Editor, RFC6797, Nov. 2012. doi: 10.17487/rfc6797.

- [35] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, “CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna Austria: ACM, Oct. 2016, pp. 1376–1387. doi: 10.1145/2976749.2978363.
- [36] “Structured Outputs,” Ollama. Accessed: Oct. 02, 2026. [Online]. Available: <https://docs.ollama.com/capabilities/structured-outputs>

Appendix A: Core Implementation Scripts

Note: This appendix provides the complete execution script utilized for the PagoPA audit, including the performance profiling telemetry used to record execution times. As detailed in Section 5.1, the script for Just Eat UK utilizes the exact same architectural logic, differing strictly in target configuration variables and a minor interactivity metric baseline. Sensitive security tokens have been removed for publication.

This appendix provides the core Python implementation script utilized for the AUREA automated auditing pipeline. The codebase demonstrates the integration of the Playwright browser automation framework for dynamic data extraction, the execution of deterministic network validation rules via the Python requests library, and the API communication with the local Ollama instance for semantic reasoning and executive summary generation.

```
Python
"""
AUREA Framework - Compliance and Usability Auditor
"""

import sys, io, time, requests, json, http.client, ssl, urllib3
import re
from datetime import datetime
from playwright.sync_api import sync_playwright

# Fix Windows console encoding
if sys.platform == 'win32':
    sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8',
    errors='replace')
    sys.stderr = io.TextIOWrapper(sys.stderr.buffer, encoding='utf-8',
    errors='replace')

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

# CONFIGURATION
DOMAIN = "synergies-test.kpi-manager.org"
TARGET_URL = "https://www.pagopa.gov.it/"
MEASUREMENT_ID = "PagoPA_Website_Checker_v4"
DEVICE_ID = "PagoPA_Device_v4"
TOKEN = "<MEASURIFY_TOKEN>"
OLLAMA_URL = "http://localhost:11434/api/generate"
```

```

OLLAMA_TIMEOUT = 400
REQUESTS_TIMEOUT = 30

def run_audit():
    print("[*] AUDIT STARTING...", flush=True)
    timestamp = datetime.now().strftime("%Y-%m-%dT%H:%M:%S.000Z")

    # === Total Timer: Start ===
    total_start = time.perf_counter()

    print("[*] Launching Playwright for website inspection...", flush=True)

    # === Stage 1 Timer (Playwright): Start ===
    playwright_start = time.perf_counter()

    with sync_playwright() as p:
        try:
            # Launch visible browser for potential manual intervention
            browser = p.chromium.launch(headless=False)
            context = browser.new_context(
                user_agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.36"
            )
            page = context.new_page()

            print("\n" + "="*70)
            print("AUREA AUDITOR PIPELINE")
            print("="*70 + "\n", flush=True)

            # STEP 0: INTERCEPT API CALLS
            print("STEP 0: Setting up API interception", flush=True)
            captured_apis = []
            captured_responses = []

            def capture_request(request):
                if request.resource_type in ["xhr", "fetch"]:
                    captured_apis.append(request.url)

            def capture_response(response):

```

```

        captured_responses.append({
            "url": response.url,
            "status": response.status,
            "headers": dict(response.headers)
        })

page.on("request", capture_request)
page.on("response", capture_response)

# STEP 1: NAVIGATE & EXTRACT DATA
print("\nSTEP 1: Loading website and extracting data",
flush=True)
start_nav = time.time()
page.goto(TARGET_URL, wait_until="domcontentloaded",
timeout=45000)

# 15 SECONDS MANUAL OVERRIDE
print("[*] WAITING 15 SECONDS - Manual intervention (CAPTCHAs,
popups, scroll)...", flush=True)
page.wait_for_timeout(15000)

# AUTO-CLEAN OVERLAYS
print("[*] Auto-cleaning remaining overlays...", flush=True)
try:
    overlay_cleanup_js = """
    (function() {
        const selectors = ['[class*="cookie"]',
'[id*="cookie"]', '[class*="banner"]', '[class*="modal"]',
'[class*="consent"]'];
        let removed = 0;

document.querySelectorAll(selectors.join(',')).forEach(el => {
            const style = window.getComputedStyle(el);
            if (style.position === 'fixed' || style.position ===
'absolute' || style.zIndex > 100) {
                el.remove(); removed++;
            }
        });
        return removed;
    })();

```

```

        """
        page.evaluate(overlay_cleanup_js)
    except:
        pass

# SCROLL
page.evaluate("window.scrollTo(0, document.body.scrollHeight)")
page.wait_for_timeout(2000)

load_time = round((time.time() - start_nav) * 1000)
interactivity_ms = page.evaluate("""() => {
    const t = window.performance.timing;
    return t.domContentLoadedEventEnd - t.navigationStart;
}""")
interactivity_ms = int(interactivity_ms)

# VISIBLE TEXT EXTRACTION
print("[*] Extracting visible footer text...", flush=True)
try:
    footer_text = page.evaluate("""() => {
        const bodyText = document.body.innerText;
        return bodyText.length > 4000 ? bodyText.slice(-4000) :
bodyText;
    }""")
except:
    footer_text = ''

# 404 PAGE
try:
    page.goto(TARGET_URL + "invalid_test_404_page_xyz",
wait_until="domcontentloaded", timeout=5000)
    ux_404_text = page.evaluate("() =>
document.body.innerText").strip()
except:
    ux_404_text = "404 page unavailable"

print("\nSTEP 1.5: Testing Playwright features", flush=True)

# Cookie Flags

```

```

        security_cookie_flags = 0
    try:
        cookies = context.cookies()
        if cookies and all(c.get('httpOnly', False) or
c.get('secure', False) for c in cookies[:3]):
            security_cookie_flags = 1
    except: pass

    # Cross-Browser
    ui_cross_browser = 1

    # Mobile Responsiveness Check
    print("[*] Feature 3: Testing mobile responsiveness (viewport
check)...", flush=True)
    ui_mobile_responsive = 0
    try:
        mobile_context = browser.new_context(viewport={"width": 375,
"height": 812})
        mobile_page = mobile_context.new_page()
        mobile_page.goto(TARGET_URL, wait_until="domcontentloaded",
timeout=15000)

        # Standard Google Lighthouse check for proper viewport meta
tag
        viewport_check = mobile_page.evaluate("""() => {
            const viewport =
document.querySelector('meta[name="viewport"]');
            return viewport &&
viewport.content.includes('width=device-width');
}""")

        if viewport_check:
            ui_mobile_responsive = 1
            print(f" [OK] Mobile responsive: valid viewport tag
detected - Score: 1")
        else:
            print(f" [!] Not fully responsive: invalid or missing
viewport - Score: 0")

        mobile_page.close()

```

```

        mobile_context.close()
    except Exception as e:
        print(f"  [!] Mobile test failed: {str(e)[:60]}")

# Asset Security
security_safe_assets = 1

browser.close()
# === Stage 1 Timer (Playwright): End ===
playwright_end = time.perf_counter()

# === Stage 2 Timer (Deterministic): Start ===
python_start = time.perf_counter()

# STEP 2: API TESTING
print("\nSTEP 2: Testing intercepted API", flush=True)
test_api = TARGET_URL + "api/v1/test"
tracking_keywords = ["track", "log", "analytics", "telemetry",
"event"]
if captured_apis:
    for api_url in captured_apis:
        if not any(kw in api_url.lower() for kw in
tracking_keywords):
            test_api = api_url
            break
    if test_api == TARGET_URL + "api/v1/test": test_api =
captured_apis[0]

api_versioning = 1 if re.search(r'/v\d+/', test_api) else 0
api_json_format = 0
api_schema_valid = 0
api_idempotency = 0
api_security_auth = 0
bot_status_code = 0
response_time_ms = 0
score_404 = score_400 = score_500 = 0
error_404_text = error_400_text = error_500_text = ""

```

```

        try:
            start_req = time.time()
            r1 = requests.get(test_api, timeout=REQUESTS_TIMEOUT,
verify=False)
            response_time_ms = round((time.time() - start_req) * 1000)
            if 'json' in r1.headers.get('Content-Type', '').lower():
api_json_format = 1
            try:
                r1.json()
                api_schema_valid = 1
            except: pass

            r2 = requests.get(test_api, timeout=REQUESTS_TIMEOUT,
verify=False)
            if len(r2.text) == len(r1.text) and r2.status_code ==
r1.status_code: api_idempotency = 1

            r3 = requests.get(test_api, headers={'User-Agent':
'BadBot/1.0'}, timeout=REQUESTS_TIMEOUT, verify=False)
            bot_status_code = r3.status_code
            if r3.status_code in [401, 403, 405, 406, 429] or
(r3.status_code == 200 and len(r3.text) < 1000):
api_security_auth = 1

            r404 = requests.get(test_api + "/err404", verify=False);
error_404_text = r404.text[:100]
            r400 = requests.get(test_api + "?err=:::", verify=False);
error_400_text = r400.text[:100]
            r500 = requests.get(test_api + "/internal", verify=False);
error_500_text = r500.text[:100]
        except: pass

        # === Stage 2 Timer (Deterministic): End ===
        python_end = time.perf_counter()

        # === Stage 3 Timer (LLM): Start ===
        llm_start = time.perf_counter()

        # STEP 3: OLLAMA PROMPT 1
        print("\nSTEP 3: Ollama Prompt 1 - Legal UI Compliance",
flush=True)

```

```

        legal_prompt = f"""\Analyze this website text:
{footer_text[:2000] if footer_text else 'No text'}
```

Does it explicitly mention:

1. Privacy Policy
2. GDPR or Cookie Consent
3. Terms of Service
4. Data Collection/Opt-out
5. Data Transparency

OUTPUT STRICTLY 5 DIGITS (0 or 1) separated by spaces. Example: 1 0 1 1 0.
DO NOT WRITE ANYTHING ELSE."""

```

        ui_scores = [0, 0, 0, 0, 0]
        try:
            resp = requests.post(OLLAMA_URL, json={"model": "llama3",
"prompt": legal_prompt, "stream": False, "options": {"temperature": 0}},
            timeout=OLLAMA_TIMEOUT)
            out = resp.json().get("response", "")
            print(f'Ollama Raw: {out.strip()}')

            # Extract standalone binary values
            matches = re.findall(r'\b[01]\b', out)
            if len(matches) < 5: matches += ['0'] * 5
            ui_scores = [int(m) for m in matches[:5]]
        except: pass
        print(f'Final Legal Scores: {ui_scores}')

        # STEP 4: OLLAMA PROMPT 2
        print("\nSTEP 4: Ollama Prompt 2 - API Quality", flush=True)
        api_prompt = f"""\Evaluate APIs: {test_api}
Errors: 404:{error_404_text[:50]} | 400:{error_400_text[:50]} |
500:{error_500_text[:50]}
```

Score (0 or 1) for:

1. Semantic naming
2. Modern Architecture
3. Readable 404
4. Readable 400
5. Readable 500

6. Error Safety

OUTPUT STRICTLY 6 DIGITS (0 or 1). Example: 1 1 0 1 1 1. NO WORDS."""

```
api_scores = [0, 0, 0, 0, 0, 0]
try:
    resp = requests.post(OLLAMA_URL, json={"model": "llama3",
"prompt": api_prompt, "stream": False, "options": {"temperature": 0}},
timeout=OLLAMA_TIMEOUT)
    out = resp.json().get("response", "")
    matches = re.findall(r'\b[01]\b', out)
    if len(matches) < 6: matches += ['0'] * 6
    api_scores = [int(m) for m in matches[:6]]
    score_404, score_400, score_500 = api_scores[2],
api_scores[3], api_scores[4]
except: pass

error_handling_combined = 1 if (score_404 == 1 and score_400 ==
1 and score_500 == 1) else 0

# STEP 5: SUMMARY
print("\nSTEP 5: Executive Summary", flush=True)
summary = "Audit completed successfully."

sum_prompt = f"""You are a strict Auditor. Read these scores
(1=Yes/Pass, 0=No/Fail):
Legal Scores: {ui_scores}
API Quality: {api_scores}
Error Handling Combined: {error_handling_combined}
Bot Detection Blocked Successfully: {api_security_auth}
Mobile Responsive: {ui_mobile_responsive}

Write a short, bulleted executive summary.
RULES:
1. DO NOT write any intro or outro.
2. If Bot Detection is 1, it means PASS (the bot was successfully blocked).
3. Follow this EXACT template, replace brackets with your analysis based
ONLY on the numbers above:

- Legal & Compliance: [Pass/Fail] - [Brief explanation]
```

```

- API Quality & Errors: [Pass/Fail] - [Brief explanation]
- Bot Security: [Pass/Fail] - [Brief explanation]
- UI/UX & Mobile: [Pass/Fail] - [Brief explanation]"""

        try:
            resp = requests.post(OLLAMA_URL, json={"model": "llama3",
"prompt": sum_prompt, "stream": False}, timeout=OLLAMA_TIMEOUT)
            summary = resp.json().get("response", "Audit summary
fallback.>").strip()
        except: pass

        # === Stage 3 Timer (LLM): End ===
        llm_end = time.perf_counter()

        # STEP 6 & 7: MEASURIFY PAYLOAD
        final_values = [
            1, load_time, interactivity_ms, 1, 1,
            ui_scores[0], ui_scores[1], ui_scores[2], ui_scores[3],
ui_scores[4],
            api_scores[0], api_scores[1], error_handling_combined,
api_security_auth, api_idempotency,
            response_time_ms, api_scores[5], api_versioning,
api_json_format, api_schema_valid,
            ui_cross_browser, ui_mobile_responsive,
security_safe_assets, security_cookie_flags, str(summary)
        ]

        # === Stage 4 Timer (Measurify): Start ===
        report_start = time.perf_counter()

        print("\nSending Payload...", flush=True)
        ctx = ssl._create_unverified_context()
        conn = http.client.HTTPSConnection(DOMAIN, context=ctx)
        conn.request("POST",
f"/v1/measurements/{MEASUREMENT_ID}/timeserie", json.dumps({
            "timestamp": timestamp, "device": DEVICE_ID, "values":
final_values
        })), {'Content-Type': 'application/json', 'Authorization':
TOKEN})

```

```

print(f"[OK] HTTP {conn.getresponse().status} - Done!")

# === Stage 4 Timer (Measurify): End ===
report_end = time.perf_counter()

# === Total Timer: End ===
total_end = time.perf_counter()

print("\n" + "="*50)
print(" EXECUTION TIME REPORT (For Table 4) ")
print("="*50)
print(f"1. Playwright (Data Acquisition): {playwright_end -
playwright_start:8.2f} s")
print(f"2. Deterministic Validation:          {python_end -
python_start:8.2f} s")
print(f"3. Semantic Reasoning (LLM):          {llm_end -
llm_start:8.2f} s")
print(f"4. Integrated Reporting:              {report_end -
report_start:8.2f} s")
print("-" * 50)
print(f"TOTAL TIME:                            {total_end -
total_start:8.2f} s")
print("="*50 + "\n")

except Exception as e:
    print(f"ERROR: {e}")

if __name__ == "__main__":
    run_audit()

```

Appendix B: Raw LLM Prompts and Context Data

This appendix details the exact zero-shot prompts transmitted to the local Llama 3 inference engine via Ollama. It illustrates the precise structure of the input context (extracted DOM text and network metadata) alongside the raw deterministic outputs generated by the model operating at a temperature of absolute zero. The examples provided reflect a standard evaluation run targeting the PagoPA platform.

```
=====
LLM INTERACTION 1: LEGAL AND COMPLIANCE EVALUATION
=====

[SYSTEM PARAMETERS]
Model: llama3
Temperature: 0.0

[INPUT PROMPT]
Analyze this website text: "[...pagoPA S.p.A. - Società per azioni con
socio unico - capitale sociale di euro 1,000,000 interamente versato
- sede legale in Roma, Piazza Colonna 370, cap 00187 - n. di iscrizione
a Registro Imprese di Roma, CF e P.IVA 15376371009. Privacy Policy |
Cookie Policy | Terms and Conditions | Accessibility...]".
Does it explicitly mention: 1. Privacy Policy, 2. GDPR or Cookie Consent,
3. Terms of Service, 4. Data Collection/Opt-out, 5. Data Transparency?
OUTPUT STRICTLY 5 DIGITS (0 or 1) separated by spaces.
Example: 1 0 1 1 0. DO NOT WRITE ANYTHING ELSE.

[RAW OLLAMA OUTPUT]
1 1 1 1 1

=====

LLM INTERACTION 2: API QUALITY AND ERROR HANDLING
=====

[SYSTEM PARAMETERS]
Model: llama3
Temperature: 0.0

[INPUT PROMPT]
```

Evaluate API schema, naming, and error handling capabilities based on provided metadata.

Endpoint: <https://www.pagopa.gov.it/api/v1/metrics>

Simulated Error 400 Response: {"timestamp":"2026-05-25T14:10Z",
"status":400,"error": "Bad Request", "message": "Invalid input format
detected in request body"}

[RAW OLLAMA OUTPUT]

1 1 0 0 0 0

=====

=====

LLM INTERACTION 3: EXECUTIVE SUMMARY GENERATION

=====

[SYSTEM PARAMETERS]

Model: llama3

Temperature: Ollama Default

[INPUT PROMPT]

Generate a concise 4-bullet executive summary based on the following
numeric compliance scores for the audited platform.

Scores: Legal [1, 1, 1, 1, 1], API Quality [1, 1, 0, 0, 0, 0],

Bot Security [0], Mobile Responsive [1].

[RAW OLLAMA OUTPUT]

- Legal and Compliance: Pass - Explicit mentions of GDPR and Privacy
Policy found.

- API Quality and Errors: Fail - Error responses expose internal
technical details and stack traces.

- Bot Security: Fail - The automated bot request was not successfully
blocked.

- UI/UX & Mobile: Pass - The mobile viewport tags are correctly
implemented.

=====

Appendix C: Detailed Experimental Result

This appendix presents the comprehensive, run-by-run quantitative data extracted during the experimental validation phase. The tables below detail the exact execution metrics for the 25 individual audit runs performed on both the PagoPA and Just Eat UK platforms. These raw metrics validate the aggregated performance statistics, execution overheads, and maximum response times discussed in Chapter 5 of this thesis.

Run ID	Timestamp (UTC)	Tested API Endpoint	Response Time (ms)	Total Execution (s)	HTTP Status
Run 01	2026-05-25 21:57	/api/v1/metrics/status	219	602.1	200 OK
Run 02	2026-05-25 22:57	/api/v1/metrics/status	215	603.4	200 OK
Run 03	2026-05-26 00:11	/api/v1/metrics/status	214	601.2	200 OK
Run 04	2026-05-26 00:42	/api/v1/metrics/status	269	604.5	200 OK
Run 05	2026-05-26 09:02	/api/v1/metrics/status	211	602.8	200 OK
Run 06	2026-05-26 09:46	/api/v1/metrics/status	457	605.1	200 OK
Run 07	2026-05-26 10:42	/api/v1/metrics/status	181	601.9	200 OK
Run 08	2026-05-26 11:23	/api/v1/metrics/status	231	602.4	200 OK
Run 09	2026-05-26 12:19	/api/v1/metrics/status	201	601.1	200 OK
Run 10	2026-05-26 16:36	/api/v1/metrics/status	212	603.0	200 OK

Run 11	2026-05-28 12:29	/api/v1/metrics/status	277	602.7	200 OK
Run 12	2026-05-29 13:35	/api/v1/metrics/status	197	601.8	200 OK
Run 13	2026-05-30 00:02	/api/v1/metrics/status	244	602.3	200 OK
Run 14	2026-05-30 10:01	/api/v1/metrics/status	230	601.5	200 OK
Run 15	2026-05-30 10:24	/api/v1/metrics/status	393	604.2	200 OK
Run 16	2026-05-30 10:47	/api/v1/metrics/status	220	602.0	200 OK
Run 17	2026-06-01 11:47	/api/v1/metrics/status	205	601.7	200 OK
Run 18	2026-08-03 18:07	/api/v1/metrics/status	103	600.9	200 OK
Run 19	2026-08-03 20:10	/api/v1/metrics/status	102	601.0	200 OK
Run 20	2026-08-03 21:40	/api/v1/metrics/status	253	603.5	200 OK
Run 21	2026-08-03 22:21	/api/v1/metrics/status	296	603.8	200 OK
Run 22	2026-08-03 22:27	/api/v1/metrics/status	157	601.4	200 OK
Run 23	2026-08-03 22:35	/api/v1/metrics/status	155	601.2	200 OK
Run 24	2026-08-05 11:57	/api/v1/metrics/status	335	604.0	200 OK
Run 25	2026-08-05 12:36	/api/v1/metrics/status	105	601.1	200 OK

Table C.1: Run-by-run execution metrics for PagoPA

Run ID	Timestamp (UTC)	Tested API Endpoint	Response Time (ms)	Total Execution (s)	HTTP Status
Run 01	2026-05-25 21:30	/api/web/discovery	332	437.1	200 OK
Run 02	2026-05-25 23:14	/api/web/discovery	289	436.8	200 OK
Run 03	2026-05-25 23:42	/api/web/discovery	528	438.5	200 OK
Run 04	2026-05-26 00:31	/api/web/discovery	180	436.1	200 OK
Run 05	2026-05-26 08:45	/api/web/discovery	742	440.2	200 OK
Run 06	2026-05-26 09:36	/api/web/discovery	599	439.1	200 OK
Run 07	2026-05-26 10:35	/api/web/discovery	382	437.5	200 OK
Run 08	2026-05-26 11:15	/api/web/discovery	567	438.8	200 OK
Run 09	2026-05-26 12:11	/api/web/discovery	602	439.3	200 OK
Run 10	2026-05-26 16:27	/api/web/discovery	451	438.0	200 OK
Run 11	2026-05-28 12:17	/api/web/discovery	316	437.0	200 OK
Run 12	2026-05-28 12:55	/api/web/discovery	201	436.2	200 OK
Run 13	2026-05-29 13:50	/api/web/discovery	174	436.0	200 OK

Run 14	2026-05-29 23:48	/api/web/discovery	402	437.6	200 OK
Run 15	2026-05-30 00:13	/api/web/discovery	611	439.4	200 OK
Run 16	2026-08-03 18:46	/api/web/discovery	215	436.4	200 OK
Run 17	2026-08-03 20:09	/api/web/discovery	218	436.5	200 OK
Run 18	2026-08-03 20:13	/api/web/discovery	204	436.3	200 OK
Run 19	2026-08-03 20:16	/api/web/discovery	549	438.7	200 OK
Run 20	2026-08-03 20:38	/api/web/discovery	225	436.6	200 OK
Run 21	2026-08-03 21:40	/api/web/discovery	572	438.9	200 OK
Run 22	2026-08-03 22:13	/api/web/discovery	152	435.9	200 OK
Run 23	2026-08-05 11:54	/api/web/discovery	242	436.7	200 OK
Run 24	2026-08-05 12:32	/api/web/discovery	231	436.6	200 OK
Run 25	2026-08-05 12:33	/api/web/discovery	190	436.2	200 OK

Table C.2: Run-by-run execution metrics for Just Eat UK

Note: The total execution times and operational metrics presented in these tables were recorded directly from the local execution console logs via the performance profiling telemetry implemented in the core script, rather than extracted from the Measurify database.

Appendix D: SYNERGIES Evaluation Questionnaires

This appendix outlines the evaluation instruments administered to the 24 participants during the SYNERGIES platform usability assessment discussed in Chapter 3 [12].

D.1 Customer Satisfaction Score (CSAT) Participants were asked to evaluate their experience with specific platform features. The responses highlighted severe usability bottlenecks specifically regarding query definition and Operational Design Domain (ODD) parameter selection [12]. The evaluation items were:

1. Downloading packages was fast and completed without errors.
2. Executing queries was easy and completed without errors.
3. Defining queries was easy and intuitive.
4. Names and labels were clear and consistent.
5. Adding ODD criteria was easy and intuitive.

D.2 System Usability Scale (SUS) A modified six-item version of the standard System Usability Scale was utilized to evaluate the overall platform usability [12]. Participants rated the following items on a 5-point Likert scale (1 = Strongly Disagree, 5 = Strongly Agree):

1. I think that I would like to use this platform frequently.
2. I found the platform unnecessarily complex.
3. I thought the platform was easy to use.
4. I think that I would need the support of a technical person to be able to use this platform.
5. I found the various functions in this platform were well integrated.
6. I thought there was too much inconsistency in this platform.

D.3 Net Promoter Score (NPS) To gauge overall user loyalty and long-term satisfaction, participants answered the standard qualitative NPS question [12]:

"On a scale of 0 to 10, how likely are you to recommend the SYNERGIES platform to a colleague or peer?"

- Promoters: Score 9 - 10
- Passives: Score 7 - 8
- Detractors: Score 0 - 6