

Article

TRACEUX: An Explainable Rule-Based Framework for Context-Aware Static UX Evaluation

Fouzia Alzhrani 

Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia; feaalzhrani@kau.edu.sa

Featured Application

TRACEUX operationalizes UX heuristics and design guidelines into rule-based static analysis workflows that support early software quality assessment. This enables developers and quality-assurance teams to detect likely UX issues, prioritize improvements based on application context, and generate structured evaluation reports before manual review or user testing.

Abstract

User experience (UX) evaluation is central to software quality, yet it remains difficult to integrate into software engineering workflows in a systematic, explainable, and early-stage manner. This paper presents TRACEUX, a framework for operationalizing UX heuristics and design guidance into a rule-based static evaluation pipeline that combines machine-interpretable formalization, executability-aware assessment, context-sensitive scoring, and actionable reporting. The framework is instantiated using Apple Human Interface Guidelines, Swift abstract syntax trees, and mobile games, and implemented in a proof-of-concept tool named TRACEHIG. Evaluation was conducted in four layers: analysis of the full rule repository, controlled synthetic validation with injected violations, baseline assessment of 12 public Swift game projects, and a case study on one project. The full repository contained 206 rules; after excluding non-iOS yet platform-specific rules, 193 rules were retained for the downstream experiments. In controlled validation, 216 injected violations yielded 99.2% precision, 61.6% recall, and an F1-score of 0.760. In baseline analysis, overall project scores ranged from 41.6 to 88.0, reflecting rule-conformance spread under the instantiated rule base rather than direct measures of UX quality. The case study demonstrated that profile-aware scoring can yield materially different UX assessments for the same codebase under different game configurations, highlighting the importance of app profiling in static UX evaluation. These findings show that a meaningful subset of UX knowledge can be operationalized into explainable, context-aware static analysis that provides structured and actionable decision support while complementing, rather than replacing, manual and empirical UX evaluation.



Academic Editor: Wonjoon Kim

Received: 24 May 2026

Revised: 22 June 2026

Accepted: 26 June 2026

Published: 6 July 2026

Copyright: © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: user experience evaluation; heuristic evaluation; game UX; static analysis; swift; abstract syntax tree; software quality

1. Introduction

User experience (UX) has become a central quality concern in modern software systems, particularly in mobile applications where interaction constraints, limited screen real

estate, context of use, and platform conventions strongly shape perceived quality [1,2]. Despite its importance, UX assessment remains difficult to integrate into software engineering workflows in a systematic and early-stage manner [3]. In practice, UX evaluation is still dominated by manual heuristic inspection, expert review, user testing, visual assessment, or post-deployment observation [4,5]. These approaches remain indispensable, but they are often resource-intensive, dependent on evaluator expertise, and more naturally applied once a prototype or mature interface is already available. Studies on mobile usability heuristics have likewise shown that mobile environments require adapted and context-sensitive evaluative criteria rather than direct reuse of generic desktop assumptions [6].

At the same time, software engineering has long relied on static analysis to reason about program properties early in the lifecycle, before runtime failures or user-visible defects emerge [7,8]. Static analysis tools are now routinely used to detect vulnerabilities, code smells, maintainability issues, and API misuse, and research has increasingly examined how such tools can support higher-level developer reasoning as well [9,10]. This raises an important question: to what extent can UX knowledge, traditionally expressed as natural-language guidance or heuristic checklists, be operationalized into analyzable rules over software artifacts?

This question is particularly relevant because a large body of UX knowledge already exists in the form of design systems, platform guidelines, and domain-specific heuristics [4]. However, such knowledge is usually written for human interpretation rather than machine-assisted evaluation. As a result, it is difficult to apply consistently across projects, difficult to scale, and weakly connected to automated quality-assurance pipelines. Prior work has shown the value of heuristic adaptation in mobile contexts [3,6], and there is early evidence that aspects of mobile usability can be assessed through code analysis [11]. Nevertheless, the literature still lacks a general framework for systematically translating UX guidance into machine-interpretable rules, linking those rules to analyzable software representations, and evaluating their practical utility through controlled validation and real-project analysis.

This paper addresses that gap through TRACEUX, a framework for operationalizing UX heuristics and design guidance into an explainable, rule-based evaluation pipeline over software artifacts. Beyond rule formalization, the framework incorporates executability-aware assessment, context-sensitive weighting, and structured decision support so that heterogeneous UX knowledge can be evaluated, scored, and reported in a principled manner. TRACEUX is designed at a level of abstraction that is not inherently restricted to a single platform, programming language, or application type. The present study instantiates this broader framework using Apple Human Interface Guidelines (HIG) as the heuristic source, Swift Abstract Syntax Trees (ASTs) as the analyzable representation, and mobile games as the demonstration domain.

To realize this instantiation, the paper introduces TRACEHIG, a proof-of-concept (PoC) implementation of TRACEUX. TRACEHIG consumes a formalized rule base, application-profile weighting data, and AST-derived facts to produce structured UX reports, and domain-level scores. The implementation is not presented as a generalized predictor of UX quality, but as a demonstrator showing that a meaningful subset of UX knowledge can be translated into explicit rules, evaluated reproducibly through static analysis and used to produce actionable feedback before more expensive manual or empirical evaluation takes place. The empirical evaluation accompanying this demonstration is therefore designed to assess feasibility and score differentiation across structurally diverse projects—not to estimate population-level properties of Swift games or to claim that observed violation rates are representative of the broader iOS development ecosystem.

The contribution of the paper is therefore twofold: conceptual and empirical. Conceptually, it proposes a framework for transforming UX guidance into machine-interpretable rule systems and for connecting those systems to static software representations. Empirically, it demonstrates this idea through a PoC instantiation and a layered evaluation involving rule executability analysis, controlled synthetic validation through injected violations, baseline analysis on public mobile game projects, and an illustrative case study.

The main contributions of this work are summarized as follows:

- A general framework for translating UX heuristics and design guidelines into machine-interpretable rules for static analysis.
- An executability-aware rule model that distinguishes fully executable, partially executable, and manual-review-only UX rules.
- A context-aware scoring and reporting model that supports weighted evaluation profiles for different application or game subtypes and produces structured, explainable feedback.
- A set of 206 formalized rules derived from Apple HIG.
- An empirical evaluation combining rule-based analysis, controlled synthetic validation, baseline analysis across real projects, and an illustrative case study on a real iOS game project.

The remainder of this paper is organized as follows. Section 2 reviews related work in UX evaluation, mobile usability heuristics, static analysis, and game UX. Material and Methods are discussed in two sections; Section 3 describes the conceptualization of the TRACEUX architecture and Section 4 details the implementation of TRACEHIG. Section 5 details the empirical and illustrative evaluations, and reports the results. Section 6 discusses implications, limitations, and threats to validity. Section 7 concludes the paper.

2. Related Work

Heuristic evaluation remains one of the most established approaches for identifying usability problems in interactive systems. General heuristic frameworks, including those associated with Nielsen and later domain-specific adaptations, are widely valued because they are structured, relatively inexpensive, and applicable early in the design process [2,12]. In the mobile context, heuristic methods continue to be used to inspect interface quality when full-scale user testing is not feasible [13]. More broadly, studies of mobile UX have emphasized that app-specific and platform-specific constraints require tailored evaluation perspectives rather than direct reuse of desktop-oriented guidelines [1,14,15].

In games, heuristic evaluation has developed into dedicated playability and player-experience frameworks. Prior work on game heuristics has shown that game UX cannot be reduced to the usability concerns of generic productivity software alone [5,16–18]. Related heuristic approaches have also been applied to domain-specific games to identify playability issues and improve interactive experience quality [19,20]. These approaches are valuable for expert inspection, but they still rely primarily on human interpretation and do not directly produce automatable software-analysis rules.

Recent UX-related work increasingly uses screenshot-based analysis and screen understanding, visual complexity measures, and app-review mining to infer UX and usability signals [21–23]. These approaches can capture properties that are not directly visible in source code, such as visual clutter, animation quality, and observed user behavior. However, they often depend on mature artifacts, deployed applications, user data, or curated training datasets. As a result, they are less suitable when developers need deterministic, explainable, and early-phase feedback derived directly from software artifacts.

Static analysis offers a complementary perspective in this space. It is already well established in software engineering for defect detection, security analysis, and maintain-

ability assessment [7,8]. In the mobile domain, prior studies have shown that abstract program representations can support higher-level inspection tasks [24,25]. Particularly, Mathur et al. [11] proposed a usability evaluation framework for mobile apps based on code analysis, suggesting that certain functional usability issues can be inferred from implementation artifacts. This is an important precedent because it shows that usability knowledge can be at least partially operationalized as code-analysis logic.

Taken together, this body of work reveals a gap between guideline-based UX inspection and automated analysis over software artifacts. Heuristic and game-heuristic approaches provide rich evaluative knowledge, but they depend primarily on expert interpretation rather than machine-interpretable rule processing [2,5,12,13,16–20]. Screenshot-, review-, and screen-understanding approaches can infer important UX signals, but they typically rely on post-implementation artifacts or observational data rather than directly analyzable software representations [21–23].

The code-analysis-based direction represented by Mathur et al. [11] is the most closely related line of work. However, prior work does not provide a general framework that integrates rule formalization, executability-aware interpretation, context-sensitive scoring, and structured actionable reporting over analyzable software representations. The present paper addresses this gap through TRACEUX and demonstrates the approach in a mobile-game setting.

3. TRACEUX Framework

3.1. Overview

TRACEUX is a general framework for translating UX heuristics and design guidance into machine-interpretable rules that can be evaluated through static analysis over software representations. The framework is designed to be independent of any single platform, programming language, or application domain. Its central premise is that a meaningful subset of UX knowledge, although commonly expressed in natural language, can be reformulated into explicit rule structures, connected to analyzable software evidence, and used to generate interpretable and actionable evaluation outputs.

At a high level, TRACEUX consists of six sequential components: (i) heuristic-source selection, (ii) rule extraction and formalization, (iii) rule executability classification, (iv) context-aware weighting, (v) score computation, and (vi) recommendation generation. The framework accepts as input a heuristic source and a software representation from which relevant facts can be extracted. It then produces as output a structured UX assessment that includes rule-level findings, domain-level scores, aggregated overall scores, and recommendation-oriented guidance. This design allows TRACEUX to serve both as an analytical framework and as a reproducible methodology for engineering-oriented UX evaluation.

Figure 1 illustrates the layered architecture of TRACEUX. The framework receives three external inputs: a heuristic source $\mathcal{H}$, a software representation S , and a context profile P . These inputs enter at different layers: $\mathcal{H}$ feeds the Knowledge Layer during rule engineering, S feeds the Evidence Layer during fact extraction, and P feeds the Scoring Layer during context-aware aggregation. The pipeline produces a structured UX evaluation report as its final output. This separation of concerns makes the framework modular: each layer can be re-instantiated independently without altering the logic of adjacent layers.

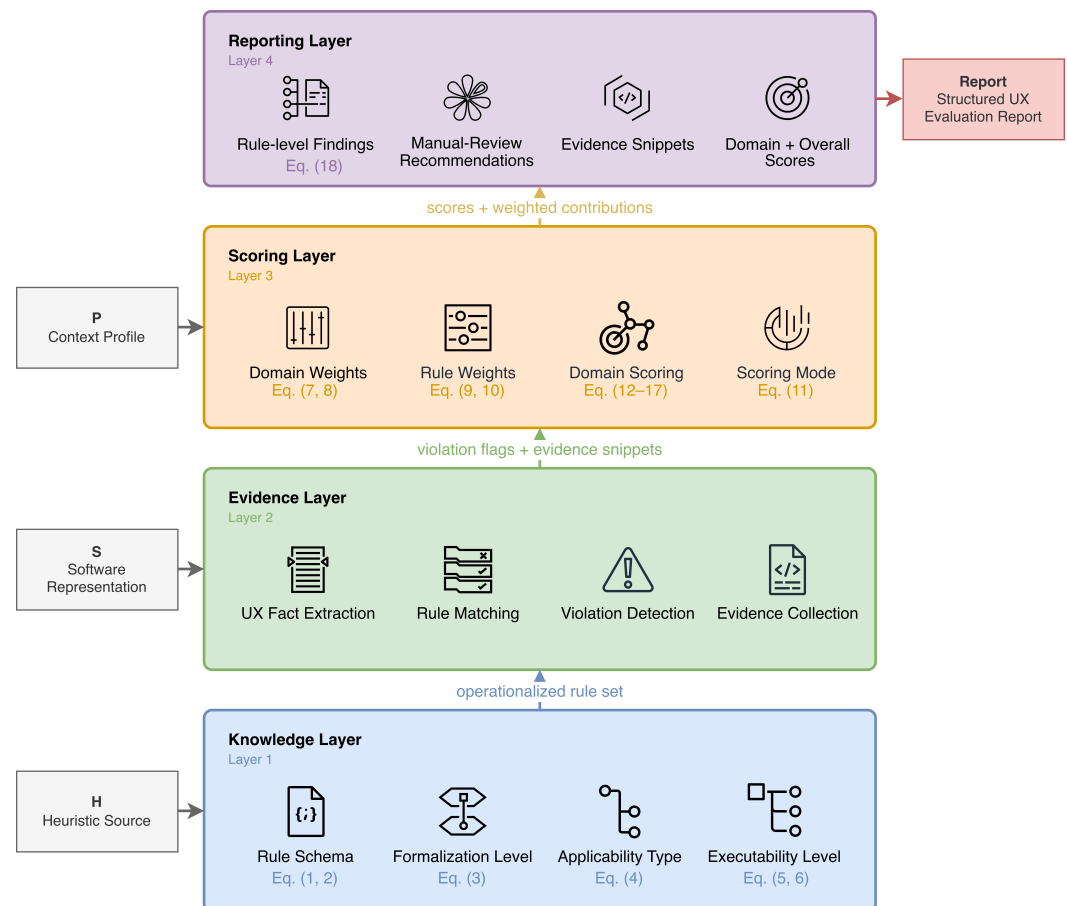


Figure 1. Layered architecture of the TRACEUX framework. Inputs $\mathcal{H}$ (heuristic source), S (software representation), and P (context profile) enter at their respective layers. The pipeline produces a structured UX evaluation report as its final output. Equation references within each layer indicate the formal definitions provided in Section 3.

To make the framework operationally explicit, Algorithm 1 summarizes the end-to-end process. The heuristic source may be a platform guideline, a domain-specific heuristic collection, or an institutional design standard. Likewise, the software representation may be an AST, a UI model, a declarative layout graph, or another structured artifact from which UX-relevant facts can be extracted. This modularity makes TRACEUX suitable for adaptation beyond the specific PoC instantiation described later in this paper.

Algorithm 1 TRACEUX Framework Workflow

Require: Heuristic source set $\mathcal{H}$, software representation S , context profile P

Ensure: Structured UX evaluation report R

- 1: Extract evaluable UX statements from $\mathcal{H}$
 - 2: Formalize extracted statements into rule set $\mathcal{R}$
 - 3: Assign executability and applicability levels and metadata to rules in $\mathcal{R}$
 - 4: Extract UX-relevant facts F from software representation S
 - 5: Match rules in $\mathcal{R}$ against facts in F
 - 6: Apply context-aware weighting using profile P
 - 7: Compute domain-level and overall scores
 - 8: Generate rule-level recommendations and structured report R
 - 9: **return** R
-

Explainability is realized at four levels in the framework. First, every rule carries traceable provenance metadata that links it to the specific heuristic guideline that motivated its formulation, so that any finding can be traced back to its authoritative source. Second,

every detected violation is accompanied by a human-readable issue message (msg_i) and a remediation-oriented recommendation (rec_i) as defined in Equation (18), so that developers receive not only a penalty signal but also a structured explanation of what was detected and what action is suggested. Third, the Executability classification (Section 3.4) makes the automation assumptions of each rule explicit: practitioners can see whether a finding comes from direct static evidence (Full), a proxy-based approximation (Partial), or a manual-review recommendation (Non-Executable). Fourth, the scoring model is fully decomposable by construction: because overall scores are computed as weighted aggregates of domain scores (Equations (12)–(16)), and domain scores are computed as weighted aggregates of individual rule contributions (Equation (11)), any overall score can be traced back to the domain-level and rule-level outcomes that produced it. Together, these properties ensure that the output of the evaluation pipeline is not opaque: every score, finding, and recommendation is grounded in explicit, human-readable reasoning.

3.2. Heuristic-to-Rule Translation

The first stage of TRACEUX is the translation of natural-language heuristic knowledge into a machine-interpretable rule base. This stage begins by identifying candidate heuristic statements that express evaluable UX expectations. A statement is considered suitable for rule extraction when it contains a distinct recommendation, prohibition, or design constraint that can be associated with an identifiable software condition, interaction pattern, or structural property.

Formally, let $\mathcal{H} = \{h_1, h_2, \dots, h_n\}$ be a set of heuristic statements extracted from a design source. The formalization function ϕ maps each heuristic statement to a rule:

$$\phi : h_i \mapsto r_i \quad (1)$$

where each rule r_i is represented as:

$$r_i = \langle id_i, title_i, dom_i, subdom_i, form_i, exec_i, app_i, ev_i, qry_i, sev_i, conf_i, rep_i \rangle \quad (2)$$

In this representation, id_i is the rule identifier, dom_i and $subdom_i$ denote the domain taxonomy, $form_i$ denotes the formalization level, $exec_i$ defines executability level, app_i captures applicability type, ev_i and qry_i define the evidence model, sev_i captures severity, $conf_i$ denotes formalization confidence, and rep_i denotes reporting and recommendation metadata. This presents the core abstract tuple capturing the minimal set of fields required for rule operationalization. The proposed schema extends this tuple with additional fields described in the Rule Schema paragraph below; the abstract tuple is preserved here for notational clarity.

The formalization stage is not merely a transcription task. It requires analytical interpretation to determine whether a heuristic statement expresses a testable condition, how broad its applicability is, what evidence may be sufficient or insufficient for assessment, and how its output should be framed when communicated to developers or evaluators. For this reason, the rule schema should preserve enough metadata to support transparent reasoning about rule outcomes.

Algorithm 2 summarizes the generic rule extraction and formalization process. A key advantage of this formulation is that it can be reused across heuristic ecosystems. Once the schema and formalization logic are defined, the same methodology can be applied to different guideline sources without changing the conceptual structure of the framework.

Algorithm 2 Generic Heuristic-to-Rule Formalization**Require:** Heuristic source set $\mathcal{H}$ **Ensure:** Formalized rule set $\mathcal{R}$

```

1: Initialize  $\mathcal{R} \leftarrow \emptyset$ 
2: for all heuristic statement  $h \in \mathcal{H}$  do
3:   if  $h$  expresses an evaluable UX expectation then
4:     Assign unique identifier and title
5:     Determine domain and subdomain
6:     Define executability and applicability conditions
7:     Specify evidence model and reporting metadata
8:     Assign severity and confidence
9:     Create rule  $r$ 
10:    Add  $r$  to  $\mathcal{R}$ 
11:   end if
12: end for
13: return  $\mathcal{R}$ 

```

Rule Schema. At an abstract level, each TRACEUX rule is represented as a structured schema consisting of: (1) identification and classification metadata, including Rule ID, Title, Domain, Subdomain, and Tags; (2) semantic metadata, including Rationale, Formalization Level, Executability, Severity, and Confidence; (3) provenance metadata linking the rule to its original guideline Source; (4) Applicability constraints specifying the contexts in which the rule is relevant; (5) one or more Query specifications for retrieving static evidence from software representation-derived facts; (6) Evidence-capture settings defining which contextual fields should be reported; (7) Reporting fields containing a human-readable issue message and recommendation; and (8) maintenance and scoring metadata, including Weights, Versioning, and Manual-review status. This design allows researchers and practitioners to reuse the rule set as-is, extend it with new heuristics, or adapt it to different programming languages by implementing alternative query backends, without modifying the conceptual UX model.

3.3. Formalization Level and Applicability Type

Two additional dimensions were explicitly encoded during rule construction: Formalization Level and Applicability. These dimensions help distinguish not only what a rule states, but also how it should be interpreted and under which circumstances it should be considered relevant.

The Formalization Level field separates *Declarative* rules from *Behavioral* rules. Declarative rules describe relatively stable properties, constraints, or design expectations, such as the use of safe areas, accessibility labels, or purpose strings. Behavioral rules instead describe interactional, temporal, or process-oriented expectations, such as when permissions should be requested, how onboarding should unfold, or how controller connection changes should be handled. This makes the rule set oriented primarily toward codifying design constraints, while still preserving a substantial subset of behavior-oriented guidance.

Let $\mathcal{F}(r_i)$ denote the formalization class of rule r_i . Then:

$$\mathcal{F}(r_i) \in \{\text{Declarative}, \text{Behavioral}\} \quad (3)$$

The Applicability field defines whether a rule should be interpreted as universally relevant or only under stated conditions. In practice, this field proved important because many UX rules are meaningful only for projects that implement specific features, target particular platforms, or expose certain interface elements. This emphasizes that the rule set should not be designed as a flat checklist, but as a context-scoped knowledge base in

which some rules become relevant only when their stated preconditions hold. *Unconditional* rules apply universally across all contexts, meaning that they can be assessed wherever the relevant code pattern appears. In contrast, *Conditional* rules are activated only when certain contextual criteria are met (e.g., platform, UI component type, or feature usage). This distinction ensures that the evaluation remains both accurate and context-aware, preventing irrelevant rule application while preserving comprehensive UX coverage.

Let $\mathcal{A}(r_i)$ denote the applicability class of rule r_i . Then:

$$\mathcal{A}(r_i) \in \{\text{Conditional}, \text{Unconditional}\} \quad (4)$$

Together, these two dimensions strengthen the semantics of the rule base. *Formalization Level* distinguishes static design constraints from behavior-oriented expectations, while applicability prevents overgeneralization by limiting each rule to the contexts for which it was intended. This also supports later layers of the framework, where executability, weighting, and reporting depend not only on the rule content itself, but also on the circumstances in which the rule is valid.

3.4. Executability Level

Not all UX heuristics can be evaluated with the same degree of automation. Some recommendations correspond closely to software-observable conditions, whereas others depend on visual interpretation, runtime behavior, user perception, or domain context that may not be recoverable from static representations alone. To address this, TRACEUX introduces an *Executability* classification that explicitly distinguishes the degree to which a rule can be operationalized. The framework uses three *Executability* classes:

- **Full:** the rule can be evaluated directly from the available software evidence with sufficient confidence for automated scoring;
- **Partial:** the rule can be evaluated only incompletely because the available evidence is suggestive but not fully conclusive;
- **Non-Executable:** the rule captures important UX guidance but cannot be reliably evaluated from the available evidence and therefore should not contribute directly to automated numeric scoring.

This classification serves two purposes. First, it prevents overclaiming the capability of static analysis. Second, it enables mixed evaluation modes in which some rules contribute quantitatively, while others remain visible as recommendation-oriented guidance for manual review.

Let $\mathcal{E}(r_i)$ denote the *Executability* class of rule r_i . Then:

$$\mathcal{E}(r_i) \in \{\text{Full}, \text{Partial}, \text{Non-Executable}\} \quad (5)$$

For scoring purposes, each *Executability* class is associated with an adjustment factor:

$$\gamma_i = \begin{cases} 1.0, & \text{if } \mathcal{E}(r_i) = \text{Full} \\ \beta, & \text{if } \mathcal{E}(r_i) = \text{Partial} \\ 0, & \text{if } \mathcal{E}(r_i) = \text{Non-Executable} \end{cases} \quad (6)$$

where $\beta \in (0, 1)$ is a discount factor for partially executable rules. This formulation makes the role of executability explicit in the evaluation model and creates a transparent boundary between automatable assessment and recommendation-only knowledge.

3.5. Context-Aware Weighting Model

UX importance is often context dependent. A design issue that is critical in one application subtype may be less central in another. TRACEUX therefore introduces context-aware weighting profiles that modulate the relative contribution of UX domains according to the evaluation context. Rather than assuming that all domains carry equal importance across all software systems, the framework allows weighting to reflect application subtype, interaction style, or domain-specific priorities.

Let $\mathcal{D} = \{d_1, d_2, \dots, d_k\}$ denote the set of UX domains. Each domain d is associated with a baseline framework weight $W_d^{(b)}$ and a context-specific profile multiplier $W_d^{(p)}$. The effective domain weight is defined as:

$$\tilde{W}_d = W_d^{(b)} \cdot W_d^{(p)} \quad (7)$$

This formulation separates stable domain importance from context-driven prioritization. The baseline weight expresses the general significance of a domain in the framework, while the profile multiplier adjusts that importance for a particular application context.

The framework also distinguishes between priority and standard domains. A domain is considered priority-weighted if its effective weight exceeds a threshold τ :

$$P_d = \begin{cases} 1, & \text{if } \tilde{W}_d \geq \tau \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

This priority distinction supports scoring strategies that place stronger emphasis on domains considered central to the target context. Because the weighting mechanism is profile-based, TRACEUX can be adapted to different application categories without modifying the underlying framework logic.

3.6. Formal Scoring Model

The scoring model of TRACEUX aggregates rule-level outcomes into domain-level and overall UX scores while preserving the effects of rule confidence, executability, and context-aware weighting. Each rule contributes a weighted amount to its associated domain, and domain scores are then aggregated into a project-level result using one of several supported scoring modes.

Let $\mathcal{R} = \{r_1, r_2, \dots, r_n\}$ denote the rule set. For each rule r_i , let w_i denote its rule weight, c_i its confidence, and $m_i \in \{0, 1\}$ its violation indicator, where $m_i = 1$ indicates that the rule is flagged as violated and $m_i = 0$ indicates no detected violation. The effective rule contribution is defined as:

$$\omega_i = w_i \cdot c_i \quad (9)$$

Incorporating executability yields the adjusted rule contribution:

$$\omega_i^* = \gamma_i \cdot \omega_i \quad (10)$$

where γ_i is given by Equation (6).

For a domain d , let $\mathcal{R}^d \subseteq \mathcal{R}$ denote the subset of rules belonging to that domain. The domain score is defined conditionally to avoid division by zero:

$$S_d = \begin{cases} 100 \times \frac{\sum_{r_i \in \mathcal{R}^d} \omega_i^* (1 - m_i)}{\sum_{r_i \in \mathcal{R}^d} \omega_i^*} & \text{if } \sum_{r_i \in \mathcal{R}^d} \omega_i^* > 0 \\ \text{undefined} & \text{if } \sum_{r_i \in \mathcal{R}^d} \omega_i^* = 0 \end{cases} \quad (11)$$

When $\sum_{r_i \in \mathcal{R}^d} \omega_i^* = 0$, the domain d is excluded from numeric aggregation. This condition arises when a domain contains only Non-Executable rules, whose adjusted contribution $\gamma_i = 0$ by Equation (6). Such domains are excluded from numeric scoring and contribute recommendation-only output.

The overall linear score is computed as the weighted average of domain scores:

$$S_{\text{linear}} = \frac{\sum_{d \in D} \tilde{W}_d S_d}{\sum_{d \in D} \tilde{W}_d} \quad (12)$$

To emphasize high-priority domains more strongly, TRACEUX also supports an exponential mode:

$$S_{\text{exp}} = \frac{\sum_{d \in D} \tilde{W}_d^\alpha S_d}{\sum_{d \in D} \tilde{W}_d^\alpha} \quad (13)$$

where $\alpha > 1$ controls the amplification effect.

Let $\mathcal{D}_p$ denote the set of priority domains and $\mathcal{D}_s$ the set of standard domains. Their aggregate scores are:

$$S_{\text{priority}} = \frac{\sum_{d \in \mathcal{D}_p} \tilde{W}_d S_d}{\sum_{d \in \mathcal{D}_p} \tilde{W}_d} \quad (14)$$

$$S_{\text{standard}} = \frac{\sum_{d \in \mathcal{D}_s} \tilde{W}_d S_d}{\sum_{d \in \mathcal{D}_s} \tilde{W}_d} \quad (15)$$

These quantities support multiple aggregation strategies, including priority-based and hybrid combinations. A weighted harmonic formulation may also be used when weak domains should be penalized more strongly:

$$S_{\text{harm}} = \frac{\sum_{d \in D} \tilde{W}_d}{\sum_{d \in D} \frac{\tilde{W}_d}{S_d + \varepsilon}} \quad (16)$$

where ε is a small positive constant applied as a smoothing term to prevent division by zero when any domain score $S_d = 0$. This situation arises for domains in which all matched rules are flagged as violated; the smoothing term ensures numerical stability while preserving the harmonic mean's sensitivity to low-scoring domains.

The hybrid scoring mode used in Section 5.2.4 combines Equations (14) and (15), as shown in Equation (17), to compute a weighted composite of the priority-domain score S_{priority} and the standard-domain score S_{standard} :

$$S_{\text{hybrid}} = \lambda \cdot S_{\text{priority}} + (1 - \lambda) \cdot S_{\text{standard}} \quad (17)$$

where $\lambda \in (0, 1)$ is a configurable blending parameter. This configuration places greater weight on priority domains while retaining the contribution of standard domains. This scoring family enables TRACEUX to support different analytical approaches without changing the underlying rule formalization or evidence-extraction logic.

3.7. Recommendation Model

A core design objective of TRACEUX is not merely to produce scores, but to produce actionable feedback. For that reason, each rule is associated with recommendation-oriented metadata that defines how the outcome of the rule should be communicated to practitioners. This makes TRACEUX a recommendation-capable evaluation framework rather than a measurement system alone.

Let rep_i denote the reporting metadata associated with rule r_i . This metadata includes at least two elements: a human-readable message describing the identified issue and a recommendation describing how the issue may be addressed. Accordingly, each rule can be associated with a recommendation tuple:

$$\Psi(r_i) = \langle msg_i, rec_i \rangle \quad (18)$$

where msg_i is the explanatory message and rec_i is the remediation-oriented recommendation.

When a rule is flagged, the framework emits the corresponding recommendation tuple. For executable rules, the recommendation accompanies the numeric effect of the detected violation. For non-executable rules, the recommendation may constitute the primary output of evaluation, since such rules are intended to support manual inspection rather than automated penalty assignment.

The recommendation model therefore plays two roles. First, it preserves the interpretability of the evaluation by linking each finding to explicit guidance. Second, it supports practical use by turning heuristic knowledge into directly actionable outputs. This is particularly important in engineering settings where evaluators and developers need more than numeric summaries; they need indications of what to inspect, what to revise, and why the reported issue matters.

4. TRACEHIG PoC Realization

4.1. Motivation for the Instantiation

To demonstrate the feasibility of TRACEUX, this study instantiates the framework in a concrete PoC realization named TRACEHIG. The purpose of this instantiation is not to define the limits of the framework, but to show that the proposed methodology can be implemented end-to-end within a realistic and demanding application setting.

Three design choices guided this realization. First, Apple HIG [26] was selected as the heuristic source because it provides a rich and well-structured body of platform-specific UX guidance. It includes interaction patterns, layout behavior, accessibility-related advice, semantic conventions, feedback expectations, and system integration practices. Second, Swift was selected as the implementation language context because it is widely used for Apple-platform development [27] and because its code can be transformed into analyzable AST representations [28]. Third, mobile games were selected as the application domain because they represent a UX-intensive software class in which interaction quality, responsiveness, input handling, onboarding, continuity, accessibility, and visual clarity are especially salient [5,17,18,29,30].

This combination provides a strong demonstration context for the framework. Apple HIG offers a sufficiently rich source of heuristic knowledge for rule extraction, Swift ASTs provide a structured software representation from which relevant facts can be extracted,

and games provide a demanding domain in which the practical value and limitations of the framework can be meaningfully observed.

4.2. Apple HIG-Based Rule Engineering

In this study, the TRACEUX framework was instantiated through a rule-engineering process centered on Apple HIG and related Apple developer documentation for game-relevant interaction and platform behavior. The source review focused on guidance applicable to Swift-based mobile games, including interaction handling, layout adaptation, feedback, accessibility, launching behavior, semantic clarity, input support, and system integration. From this review, prescriptive statements were extracted and transformed into a structured rule base with domain, subdomain, severity, confidence, applicability, evidence, and recommendation metadata.

Then, each formalized rule was assigned an Executability class based on whether Swift AST evidence was sufficient to support automated assessment. Rules were labeled as *Full* when their conditions could be directly checked from AST-visible evidence, *Partial* when AST evidence provided only incomplete or suggestive support, and *Non-Executable* when the underlying guideline remained important but could not be assessed reliably from static structure alone. This classification was used later to control rule participation in automated scoring.

4.3. Game Profiling

4.3.1. Profile Categories

To support context-sensitive evaluation, three game profiles were defined: *action/arcade*, *puzzle*, and *general*. These profiles were chosen to provide meaningful differentiation across the study corpus while avoiding overly fine-grained genre distinctions that would be difficult to assign consistently in a small and heterogeneous set of public game repositories. The design rationale for each profile draws on established literature on game UX and mobile usability [4,17,30]. The action/arcade profile elevates the Interaction, Input Support, Feedback, and Lifecycle domains, because fast-paced gameplay is failure-sensitive to input latency, control responsiveness, and interruption handling—concerns well-documented in the game heuristics literature. The puzzle profile instead places greater multiplier weight on Layout, Semantics, Adaptability, and Onboarding, reflecting the primacy of clarity, readability, and low-friction interaction in goal-oriented, cognitive-load-sensitive contexts. The general profile applies equal domain weights and serves as a neutral, uncontextualized baseline for projects that do not fit primarily into the other categories.

4.3.2. Profile Multipliers

Profile multipliers were manually specified as domain-level adjustment factors layered over baseline domain weights. The intent was not to learn optimal weights statistically, but to encode plausible differences in UX emphasis across broad game categories in a transparent and reproducible way. We emphasize that these multipliers are preliminary, transparently specified settings rather than empirically validated weights: they were not calibrated against expert ratings, inter-rater agreement, or statistical optimization. They should therefore be read as an initial design choice whose robustness to alternative weightings is examined in the perturbation analysis reported below (Table 1).

To assess whether the overall project scores are sensitive to the preliminary weight choices, a simple perturbation analysis was conducted on the baseline corpus of 12 projects. The effective domain weight for each domain is the product of the baseline weight $W_d^{(b)}$ and the profile multiplier $W_d^{(p)}$ (Equation (7)). Perturbation was applied by scaling all effective weights toward or away from the neutral value of 1.0 by a factor of 0.5 (flattening) and 1.5

(steepening), and by replacing all weights with a uniform value of 1.0 (equal weighting). Table 1 reports the overall scores under each scenario.

Across ten of the 12 projects, the maximum score change under any perturbation scenario ranged from 1.98 to 10.78 points, with a mean of 7.0 points, and rank ordering was fully preserved in all cases. Two projects—AR-Multipendulum and Frozen Pengu—showed larger sensitivity (approximately 36 points under equal weighting). This behavior does not arise from weight sensitivity per se, but from the structure of the hybrid scoring formula: both projects record zero violations in all high-priority domains, yielding a priority score of 100.0, which the hybrid formula blends at 75% weight (Equation (17)).

When weights are equalized, the priority/standard distinction collapses, merging these components and producing the observed drop. This behavior confirms that the weight scheme correctly rewards conformance in high-priority domains rather than indicating instability of the preliminary weights. Across all projects, the bottom rank (Potato Game, 41.6) and the top two ranks (AR-Multipendulum, Frozen Pengu) were preserved under all perturbation scenarios, indicating that the qualitative ordering and practical interpretation of results are robust to the current weight specification.

Table 1. Weight perturbation sensitivity analysis on baseline project scores. Flat = weights scaled 50% toward 1.0; Steep = weights scaled 50% away from 1.0; Equal = all weights set to 1.0. Baseline scores use the hybrid formula (Equation 17); perturbation columns use linear domain-score aggregation as a sensitivity approximation.

Project	Baseline	Flat	Steep	Equal	Max Δ
Flappy	62.09	54.18	54.94	53.74	8.35
AR-Multipendulum	88.00	51.99	51.99	51.99	36.01 *
Swift-2048	52.58	50.98	50.60	51.20	1.98
Frozen Pengu	87.90	51.61	51.61	51.61	36.29 *
Snake Classic	61.56	53.52	54.29	53.09	8.47
Potato Game	41.61	38.10	38.19	38.04	3.57
Rock Paper Scissors	69.01	58.56	59.13	58.23	10.78
Hyper Snake	62.99	54.63	55.42	54.17	8.82
Spring Festival Crush	50.97	48.39	48.23	48.49	2.74
Hoop	59.46	51.96	52.72	51.52	7.94
Physics Vehicle	63.28	54.54	55.37	54.06	9.22
Flappy Canary	61.99	53.94	54.71	53.51	8.48

* Large Δ reflects hybrid formula structure, not weight instability.

4.4. Technical Implementation

TRACEHIG [31] was implemented as a Python-based evaluator operating over file-based inputs v.3.9.6. The implementation consumes four primary JSON artifacts: a formalized rule base, a UX configuration file containing baseline domain weights and scoring parameters, a game-profile configuration file containing profile-specific domain multipliers, and a Swift AST file representing the target project.

AST generation was handled externally using a Swift parsing pipeline that exports source structure into JSON form [32]. The evaluator assumes a normalized AST representation that preserves node type, source location, raw or trimmed code text, and contextual information such as enclosing type and enclosing function when available. An internal adapter traverses this representation and extracts UX-relevant facts used during rule evaluation. Each extracted fact contains fields such as file, line, column, node type, snippet, enclosing type, enclosing function, raw text, identifiers, called expressions, and module names. This structured representation enables rule evaluation through pattern matching rather than direct dependence on raw source text alone.

The implementation was organized into four main components: fact extraction, rule matching, scoring, and reporting. The fact-extraction component identifies relevant

AST elements such as imports, declarations, identifiers, member accesses, and function calls. The rule-matching component evaluates the formalized evidence conditions associated with each rule against the extracted fact set. The scoring component computes domain-level and overall scores using the selected aggregation mode, while incorporating rule confidence, executability adjustment, baseline domain weights, and profile multipliers. The reporting component emits a structured JSON report that includes overall and domain scores, severity-distributed findings, evidence snippets, and manual-review recommendation items.

To support comparative experimentation, TRACEHIG implements five scoring modes: linear, exponential, priority-based, weighted harmonic, and hybrid. This allowed the same rule and evidence base to be analyzed under alternative aggregation assumptions without changing the underlying rule-engineering pipeline. Overall, TRACEHIG demonstrates that the abstract TRACEUX framework can be realized as a working pipeline from guideline-derived rules and AST-based evidence to interpretable UX evaluation artifacts.

5. Evaluation and Results

This section reports the empirical evaluation of TRACEHIG introduced in Section 4. The evaluation has two complementary aims. First, it examines the operational characteristics of the instantiated framework through rule executability coverage, controlled validation, and multi-project baseline analysis. Second, it illustrates how the resulting scoring and reporting outputs can be interpreted for a single project in an end-to-end usage scenario.

5.1. Dataset

The empirical study uses a corpus of 12 public Swift game projects, introduced in Table 2. The same corpus is used for both the controlled synthetic validation in Layer 2 and for the baseline analysis in Layer 3. All projects are evaluated with the same 193-rule Apple HIG-derived rule base, while score aggregation is performed using the same scoring implementation and project-specific game profile assignment. Game type was assigned by static-structure-based genre inference from AST evidence, using signals such as engine/framework imports, scene/control architecture, physics and collision mechanics, touch interaction style, scoring loops, and gameplay vocabulary. The final labels were manually reviewed for ambiguous cases.

Table 2. Baseline corpus used in the evaluation.

Project	Game Type
Flappy [33], Snake Classic [34], Potato Game [35], Hyper Snake [36], Flappy Canary [37], Physics Vehicle [38], Hoop [39]	Action
Swift-2048 [40], Rock Paper Scissors [41], Spring Festival Crush [42]	Puzzle
Frozen Pengu [43], AR-Multipendulum [44]	General

The corpus was selected to provide variation in the structure of the project and potential UX findings rather than to estimate the population-level properties of Swift games. Consequently, the baseline analysis should be read as a multi-project demonstration of score differentiation and reporting behavior under a common rule base, not as a claim about the prevalence of specific UX problems in the broader ecosystem.

5.2. Empirical Evaluation

5.2.1. Experimental Design

The evaluation addresses three empirical questions. First, how much of the Apple HIG-derived rule base can be operationalized? Second, when rule-relevant perturbations

are injected into project ASTs under controlled conditions, to what extent can TRACEHIG recover them? Third, when applied to unmodified real projects, does the instantiated evaluator produce differentiated and interpretable project-level and domain-level outputs?

The controlled validation follows a four-stage workflow. Starting from 12 baseline ASTs and their baseline outputs, the process first generates an injection plan by locating valid AST anchors for synthetic rule-specific perturbations. Next, the planned perturbations are injected into the AST JSON files. TRACEHIG is then executed on the modified ASTs, and the resulting detections are matched against the injection ground truth to compute precision, recall, and F1-score. The evaluation workflow therefore combines direct rule-base characterization, controlled recovery testing, and corpus-level baseline analysis within a single experimental setting.

5.2.2. Layer 1: Rule-Base Structural Analysis

The first evaluation layer examined the instantiated TRACEUX rule base itself as a structured knowledge artifact, before testing detection behavior on injected or real projects. Layer 1 first analyzed the full instantiated rule repository, which contained 206 rules, comprising 5 rules classified as *Full*, 149 as *Partial*, and 52 as *Non-Executable*. Because Layers 2 and 3 targeted iOS Swift projects, non-iOS yet platform-specific rules were then excluded from the full repository, yielding an experimental subset of 193 rules for downstream validation and baseline analysis.

Within this experimental subset of 193 rules, 5 rules were classified as *Full*, 139 as *Partial*, and 49 as *Non-Executable*, corresponding to 2.6%, 72.0%, and 25.4%, respectively. This distribution indicates that the repository is dominated by proxy-based operationalizations rather than direct static verification. This pattern is expected for UX guidance derived from Apple HIG and related platform guidance, because many recommendations refer to visual quality, runtime interaction, contextual appropriateness, perceptual accessibility, or experiential factors that cannot be resolved from AST evidence alone.

Analysis of Formalization Level showed that 144 rules were *Declarative* and 49 were *Behavioral*. This indicates that the repository primarily captures structural and design-oriented expectations, while still retaining a meaningful subset of rules that describe temporal, interactional, or process-oriented behavior. The distribution also aligns with executability patterns: all fully executable rules were declarative, while behavioral rules were almost entirely partially executable. This suggests that behavior-oriented guidance can often be approximated through static indicators, but is less likely to admit complete static verification.

Analysis of Applicability further showed that 154 rules were *Conditional* and 39 were *Unconditional*. This confirms that the rule base is predominantly context-scoped rather than universal. In other words, most rules are intended to apply only when specific interface elements, platform features, or user-facing capabilities are present. This design choice is important because it avoids treating the repository as a flat checklist and instead models UX guidance as contingent knowledge whose relevance depends on the characteristics of the target software.

The interaction between Applicability and Executability is also revealing. Of the 39 *Unconditional* rules in the experimental set, 94.87% were non-executable and 5.13% were fully executable, whereas the large majority of *Conditional* rules were partially executable. This means that rules expressing broad experiential or perceptual principles tend to remain manual-review guidance, while rules tied to specific UI constructs, APIs, or configuration patterns are more amenable to static operationalization.

The schema analysis also highlights the repository's operational granularity. Rules contained an average of 1.23 static queries, with a median of 1 and a maximum of 3,

indicating that most rules are intentionally concise rather than relying on large query bundles. Applicability conditions averaged 1.62 per rule, with most rules specifying one or two conditions, which further supports the interpretation that context is encoded explicitly but in a controlled form. In addition, 188 of the 193 rules were marked for manual review, showing that even executable rules were designed to support explainable human interpretation rather than fully autonomous judgment.

Taken together, the Layer 1 results characterize the TRACEUX repository as an executability-aware, context-scoped, and reporting-oriented knowledge base. Its structure reflects a deliberate balance: preserving broad UX coverage, operationalizing those parts that can be grounded in static evidence, and retaining the remainder as structured recommendation-oriented guidance for later reporting layers rather than excluding them from the evaluation pipeline. Layer 1 therefore evaluates the instantiated rule base not merely in terms of automation coverage, but as a structured knowledge artifact whose semantics depend on executability, formalization style, contextual applicability, and reporting design.

5.2.3. Layer 2: Controlled Synthetic Validation

The controlled validation began with 492 planned injection opportunities across the 12-project corpus. Due to the structural sparsity of game code (i.e., absence of Dynamic Type calls, safe-area constraints, and scene lifecycle hooks), 276 of 492 planned injections were excluded at the planning stage because no suitable AST anchor node was present in the target files; these cases were never attempted. Thus, 43.9% of planned opportunities were actionable, while 56.1% could not be instantiated under the current AST-level injection constraints. All 216 actionable injections were successfully applied, yielding an injection execution rate of 100.0% over actionable cases and 43.9% over all planned opportunities.

The number of applied injections varied across projects. The smallest injected count was 3 in Rock Paper Scissors, whereas the largest was 32 in Potato Game. Across the corpus, the mean number of applied injections was 18.0 per project. The median was 15.0, indicating a moderate skew toward projects with larger numbers of viable injection anchors.

After subtracting baseline detections already present in the original projects, TRACE-HIG produced 799 new detections on the injected corpus, while 4059 baseline detections were suppressed. Matching the post-subtraction detections against the injection ground truth yielded 133 true positives, 1 false positive, and 83 false negatives. These values correspond to a precision of 99.2%, recall of 61.6%, and F1-score of 0.760.

Two additional ratios are useful for interpreting this result. First, the false discovery rate was 0.8% (1/134), confirming that spurious matched detections were rare in this experiment. Second, the miss rate was 38.4% (83/216), indicating that a substantial minority of injected cases remained unrecovered. The validation outcome is therefore asymmetric: the instantiated detector behaved conservatively, with very high precision but only moderate recall.

It is important to distinguish two conceptually different sources of recall limitation in this evaluation. The first concerns injections that were planned but could not be executed because no suitable AST anchor node was available in the target file; in this study, 276 of 492 planned injections fell into this category and were excluded before evaluation began. The second concerns injections that were successfully applied but subsequently went undetected by TraceHIG; all 83 false negatives belong exclusively to this second category. The injection execution rate over actionable cases was 100%: every injection that was attempted was successfully placed. The false negatives therefore reflect detection gaps in the current rule matchers, not anchor availability failures.

Analysis of the 83 false negatives reveals three recurring causes. The most frequent involved SwiftUI-style patterns not yet covered by the detector's rule matchers, which are tuned primarily toward UIKit idioms. Rule GC-001 (Interaction) was successfully injected as `Button(action: jump) { Image(systemName: "arrow.up") }.frame(width: 20, height: 20)` across multiple projects, but the detector failed to flag it in 20 of the Interaction injection cases, accounting for the largest share of FNs in that domain. Rule IOS-002 (Adaptability) contributed 10 false negatives when `.foregroundColor(.red)` was injected as a `MemberAccessExprSyntax` node, as the current matcher does not cover this SwiftUI color-modifier pattern.

A second cause was semantic indistinguishability: rule D4G-007 (Input Support) injected `import UIKit` into files that already contained that import, making the injected node functionally identical to existing code and therefore correctly suppressed by the tool's baseline deduplication—accounting for 5 of the Input Support false negatives. A third cause applied to string-literal injections lacking surrounding call context: rules GC-004 (Visual Design) and LAUNCH-002 (Launching) injected fragments such as `Image("controller-icon")` and `"LaunchScreen"` as bare string literals, which the detector cannot confidently attribute to a guideline violation without the enclosing call structure. Together, these findings indicate that the current recall gap stems primarily from incomplete SwiftUI pattern coverage rather than from fundamental detection failures, and identify a concrete development priority for the next iteration of TraceHIG.

The 276 uninstantiable cases were not uniformly distributed across domains. Accessibility (47), Typography (35), Interaction (34), Onboarding (24), and Layout (23) accounted for the largest shares of excluded cases, reflecting the structural sparsity of these concerns in small open-source game projects—for instance, the absence of Dynamic Type calls, safe-area constraints, and scene lifecycle hooks. The domain-level recall figures in Table 3 should therefore be read in light of this coverage asymmetry: low recall in a domain may reflect detection gaps, limited injection coverage, or both.

The domain-level results were uneven, as shown in Table 3. Accessibility yielded the largest number of true positives (44) and the highest recall among the larger domains (89.8%). Interaction and Typography also produced substantial recovery, with 30 and 25 true positives, respectively. By contrast, Adaptability and Layout showed very high precision but low recall, suggesting that their current rule formulations detect only a subset of eligible perturbations.

Several domains achieved perfect scores within small injected subsets, including Content Design, Settings, and Windows. However, the zero-recall outcomes for Apple Services and Privacy indicate that successful recovery was not uniform across all domains represented in the injection set. Because some of these domains are represented by only a few injected cases, the corresponding results should be interpreted as indicative rather than conclusive.

Table 3. Layer 2 detection validity by domain.

Domain	Precision (%)	Recall (%)	F1	TP	FP	FN
Apple Services	0.0	0.0	0.000	0	0	1
Privacy	0.0	0.0	0.000	0	0	7
Adaptability	100.0	9.1	0.167	1	0	10
Layout	100.0	15.4	0.267	2	0	11
Semantics	100.0	33.3	0.500	1	0	2
Launching	100.0	40.0	0.571	2	0	3
Visual Design	100.0	45.5	0.625	5	0	6
In App Purchase	100.0	50.0	0.667	1	0	1
Input Support	100.0	58.3	0.737	7	0	5
Interaction	100.0	60.0	0.750	30	0	20
Typography	96.2	67.6	0.794	25	1	12
Accessibility	100.0	89.8	0.946	44	0	5
Content Design	100.0	100.0	1.000	10	0	0
Settings	100.0	100.0	1.000	3	0	0
Windows	100.0	100.0	1.000	2	0	0

5.2.4. Layer 3: Real-Project Baseline Analysis

The baseline analysis applied TRACEHIG to the 12 original, unmodified projects. Each project was evaluated against the same 193-rule experimental rule base using hybrid score aggregation. Table 4 reports the resulting project-level results.

Table 4. Layer 3 baseline results across the 12-project corpus.

Project	Scoring Mode	Violations	Overall Score *	Priority Score *
Flappy	Hybrid	90	62.1	67.0
AR-Multipendulum	Hybrid	96	88.0	100.0
Swift-2048	Hybrid	101	52.6	54.0
Frozen Pengu	Hybrid	100	87.9	100.0
Snake Classic	Hybrid	94	61.6	66.6
Potato Game	Hybrid	121	41.6	43.9
Rock Paper Scissors	Hybrid	78	69.0	75.7
Hyper Snake	Hybrid	89	63.0	68.2
Spring Festival Crush	Hybrid	111	51.0	52.8
Hoop	Hybrid	101	59.5	64.1
Physics Vehicle	Hybrid	89	63.3	68.7
Flappy Canary	Hybrid	91	62.0	67.0

* The Overall and Priority scores measure rule conformance under the current static evidence model, which relies predominantly on proxy-based partial rules; they should not be read as direct measures of UX quality.

The baseline results exhibited substantial spread across the corpus. The overall scores ranged from 41.6 to 88.0, a span of 46.4 points. The mean overall score was 63.5, and the median was 62.1. Violations ranged from 78 to 121, with a mean of 96.7 and a median of 95.0. These values indicate that the instantiated evaluator did not collapse the corpus into a narrow score interval; instead, it produced discernible separation across projects under a common rule base and scoring mode.

The highest overall scores were observed for AR-Multipendulum (88.0) and Frozen Pengu (87.9), followed by Rock Paper Scissors (69.0). The lowest scores were observed for Potato Game (41.6), Spring Festival Crush (51.0), and Swift-2048 (52.6). The corresponding ordering by violation count was broadly consistent, although not perfectly monotonic, indicating that the final score reflects more than raw violation totals alone.

At the corpus level, the reported weakest recurring domains were Semantics, Content Design, Launching, Input Support, Compatibility, Windows, UI Integration, and Assets, each with a mean score of 0.0 across the 12 projects. Their aggregate violation counts were

24, 48, 12, 96, 12, 24, 12, and 24, respectively. Normalized per project, these correspond to average violation rates of 2.0, 4.0, 1.0, 8.0, 1.0, 2.0, 1.0, and 2.0 per project. Among these domains, Input Support was the most recurrent in absolute terms, accounting for 96 violations across the corpus.

It is important to interpret these scores with appropriate caution regarding what they measure. Because Layer 3 does not use labeled ground truth, the values reported in Table 4 are *rule-conformance scores*. They quantify the degree to which each project satisfies the operationalized subset of Apple HIG-derived rules under the current static evidence model. They do not constitute direct measures of UX quality, user satisfaction, perceived usability, or interaction effectiveness.

A project may score highly because its codebase exhibits few AST-visible guideline violations while still providing a poor UX due to factors that are not recoverable from static representations alone, such as visual clarity, animation fluency, interaction responsiveness, or subjective engagement. The inverse is equally possible: a low-scoring project may in practice offer a satisfactory experience if its violations are minor or compensated by runtime behavior. The distinction between rule conformance and UX quality is further discussed in the context of construct validity in Section 6.3.

5.3. Illustrative Case Study

5.3.1. Experimental Design

To examine the effect of app profiling and score aggregation in TRACEUX, we evaluated Frozen Penguin using a 3×5 design that combines three app profiles (General, Action, and Puzzle) with five Scoring Modes (Linear, Exponential, Weighted Harmonic, Priority, and Hybrid). The weight exponent for exponential/hybrid modes defaults to 1.5, while the weight threshold for priority domains defaults to 1.2.

Frozen Penguin was selected as a suitable case because its report contains a diverse mixture of violations spanning multiple domains. This makes it appropriate to examine whether different contextual assumptions emphasize different weaknesses rather than merely changing a single overall score. Moreover, the project represents a 2D game setting, making it useful for studying how alternative profile interpretations influence the weighting of domains and gameplay-related concerns. This choice was especially appropriate because Frozen Penguin is described by its developers as including both puzzle and arcade modes, making profile variation consistent with the game's own intended design space.

5.3.2. Case Analysis

The same codebase was analyzed in all 15 runs, with only the evaluation configuration varied; accordingly, every run inspected 193 rules and produced the same violation log of 100 violations, 149 recommendations, 2 critical violations, 35 high violations, 48 medium violations, 15 low violations, and 49 manual review items.

This invariance is important because it shows that score differences do not arise from unstable detection. Instead, Frozen Penguin serves as a controlled case that isolates the interpretive role of TRACEUX: the framework holds the detected UX evidence constant and changes only how that evidence is weighted and aggregated under different application profiles.

The General profile and Linear scoring together provide the natural baseline for this analysis. In the General profile, all domains have equal weight of 1.0, while Linear scoring aggregates domain performance without the additional priority-sensitive or harmonic penalties used in the other modes; thus, the General/Linear result of 51.61 represents the unprofiled baseline assessment of the codebase.

Against this baseline, Frozen Penguin exhibits a sharply uneven UX structure. Several domains score 100.0, including Gameplay, Progression, Competition, Multiplayer, UX, Inclusion, Information Architecture, and Performance, whereas multiple other domains are very weak or completely failing, including Interaction at 16.82, Feedback at 11.93, Layout at 26.66, Adaptability at 28.93, Onboarding at 33.96, and Semantics, Content Design, Settings, Input Support, Launching, Compatibility, Windows, UI Integration, Lifecycle, Launch Experience, and Assets at 0.0.

This polarized structure explains why a single average-like baseline score is insufficient. Frozen Penguin is not uniformly poor; rather, it combines strong support for some gameplay- and infrastructure-oriented concerns with concentrated weaknesses in interaction design, settings, semantic correctness, onboarding, and multi-input support, which are precisely the kinds of imbalances that app profiling is intended to expose.

The effect of app profiling becomes clearest when the baseline is compared with the Action and Puzzle profiles. Under Linear scoring, the three profiles remain close, with scores of 51.61 for General, 52.45 for Action, and 51.68 for Puzzle, indicating that simple weighted averaging alone does not materially change the interpretation of the codebase.

However, once profile-sensitive scoring is introduced, the differences become substantial. Under Priority scoring, Frozen Penguin scores 85.48 in General, 59.14 in Action, and 56.08 in Puzzle, while under Hybrid scoring it scores 87.90, 60.16, and 56.55, respectively; this shows that app profiling becomes a major source of variation when the aggregation explicitly distinguishes between priority and non-priority domains.

These results are analytically revealing. Because the General profile is the unprofiled baseline with uniform weights, it does not foreground genre-specific risk areas; by contrast, the Action and Puzzle profiles assign greater importance to domains that are more consequential for those gameplay styles, and Frozen Penguin performs poorly in several of those domains, particularly Interaction, Feedback, Layout, Input Support, Content Design, Adaptability, and Onboarding.

The sub-score decomposition further clarifies this behavior. In General/Priority, the overall score is 85.48 with `priority_score` = 100.0 and `standard_score` = 51.61, whereas Action/Priority yields 59.14 with `priority_score` = 65.1 and `standard_score` = 45.24, and Puzzle/Priority yields 56.08 with `priority_score` = 59.94 and `standard_score` = 47.08; a similar pattern holds for Hybrid, where the General profile again reports `priority_score` = 100.0, compared with 65.21 for Action and 59.82 for Puzzle. Figure 2 illustrates this effect by showing that the severity counts are unchanged across profiles, whereas the overall score, sub-score decomposition, and weighted domain contributions vary substantially.

This confirms the importance of app profiling in TRACEUX. The same defect set appears substantially better when evaluated through a generic, non-profiled configuration than when assessed through profiles that emphasize genre-relevant UX demands, indicating that app profiling is not a cosmetic parameter but a necessary contextual layer for meaningful interpretation. Figures 2–4 illustrate the effect of app profiling on scoring.

The scoring modes also contribute distinct analytical perspectives. Exponential scoring remains close to the Linear baseline, yielding 51.61 in General, 52.93 in Action, and 51.74 in Puzzle, which suggests that for Frozen Penguin the additional penalty curvature has limited influence at the overall level. Weighted Harmonic, in contrast, is consistently the most conservative mode, producing 49.00 in General, 47.12 in Action, and 48.42 in Puzzle. This is expected because harmonic aggregation is strongly affected by low and zero domain scores, making it particularly effective at preventing fully failing domains such as Settings and Input Support from being masked by domains that score 100.0.

At the rule level, two critical violations persist across all runs: BTN-005 in the Semantics domain and PRIV-002 in the Privacy domain. Their recurrence across every profile and

scoring mode shows that some findings in TRACEHIG are configuration-invariant and should be interpreted as foundational UX risks rather than profile-dependent concerns.

Table 5 presents representative high-severity findings identified in the Frozen Pengu case study. The selected examples indicate that the most consequential issues were distributed across multiple UX dimensions, including accessibility, interaction design, input support, and platform-aware usability, rather than being concentrated in a single area. Collectively, the findings point to recurring concerns related to audio accessibility, target size adequacy, control sizing, and reliance on limited interaction methods. The associated recommendations further illustrate the decision-support role of TRACEHIG by linking each issue to a concrete corrective direction grounded in platform-aligned UX guidance. These examples show how the framework can translate static-analysis evidence into structured and actionable feedback at the code level.

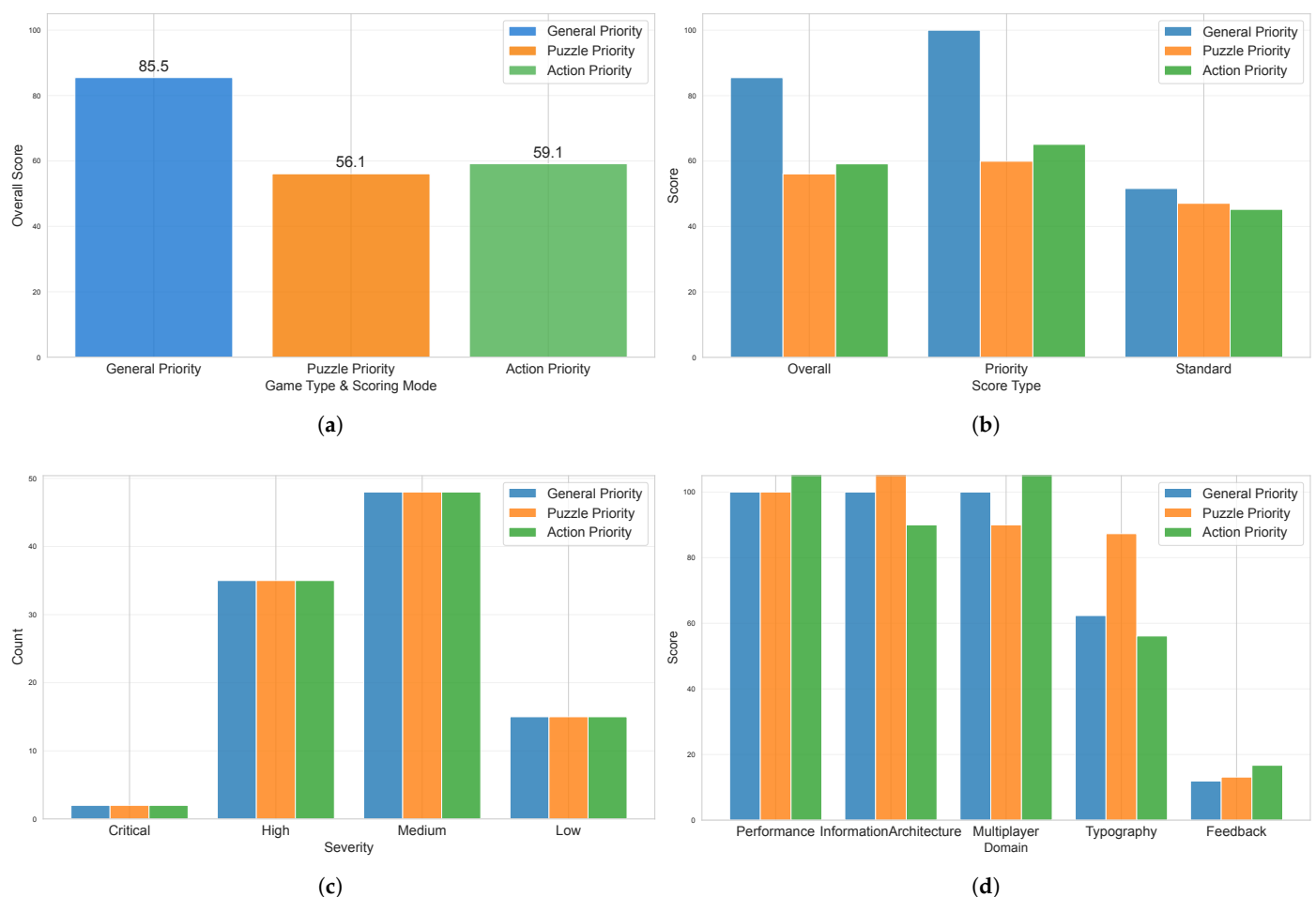


Figure 2. Frozen Pengu results under the Priority scoring model for the General, Action, and Puzzle profiles. Although the underlying violation counts are unchanged, the overall score, priority/standard breakdown, and weighted domain contributions vary across profiles, showing the effect of app-specific prioritization in TRACEUX. (a) Overall Score Comparison. (b) Score Breakdown Comparison. (c) Violations by Severity Comparison. (d) Weighted Domain Score Comparison (Top 5 Most Different).

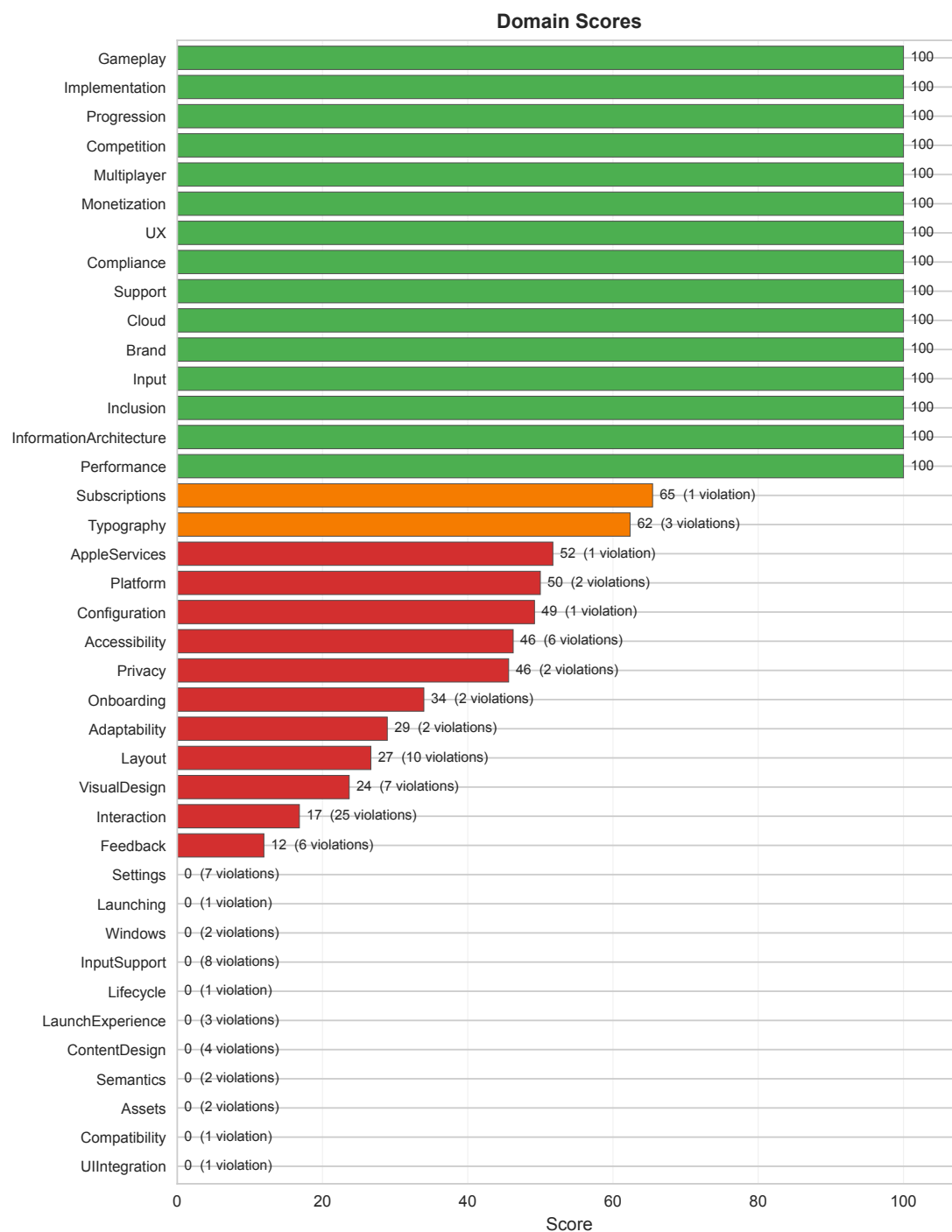


Figure 3. Domain scores for Frozen Penguin across all evaluated UX domains, sorted from lowest to highest; values in parentheses report the number of violations detected in each domain. Because the same codebase and violation log underlie every run, these scores are invariant across app profiles and scoring modes—only their weighting differs (see Figure 4).

A notable outcome of this case study is that the profile-based interpretation produced by TRACEHIG is consistent with the developer’s own description of Frozen Penguin as a game that combines puzzle and arcade play. Rather than yielding an arbitrary profile assignment, the framework produced differentiated results that remain plausible for a hybrid application and that show somewhat closer alignment with Puzzle-oriented UX demands than with Action-oriented ones. This agreement suggests that the profiling

mechanism captures meaningful properties of the application rather than introducing artificial variation.

Overall, the Frozen Pengu case study demonstrates two properties of TRACEUX. First, the framework is stable at the detection layer, since the same codebase yields the same underlying rule violations in every run; second, it is meaningfully adaptive at the valuation layer, since the reported UX quality changes substantially once the baseline configuration is replaced by profile-aware evaluation. Most importantly, the case study shows why the General/Linear setting should be treated only as a baseline. It provides a useful reference point for the unprofiled codebase, but it can understate the practical significance of genre-relevant UX weaknesses.

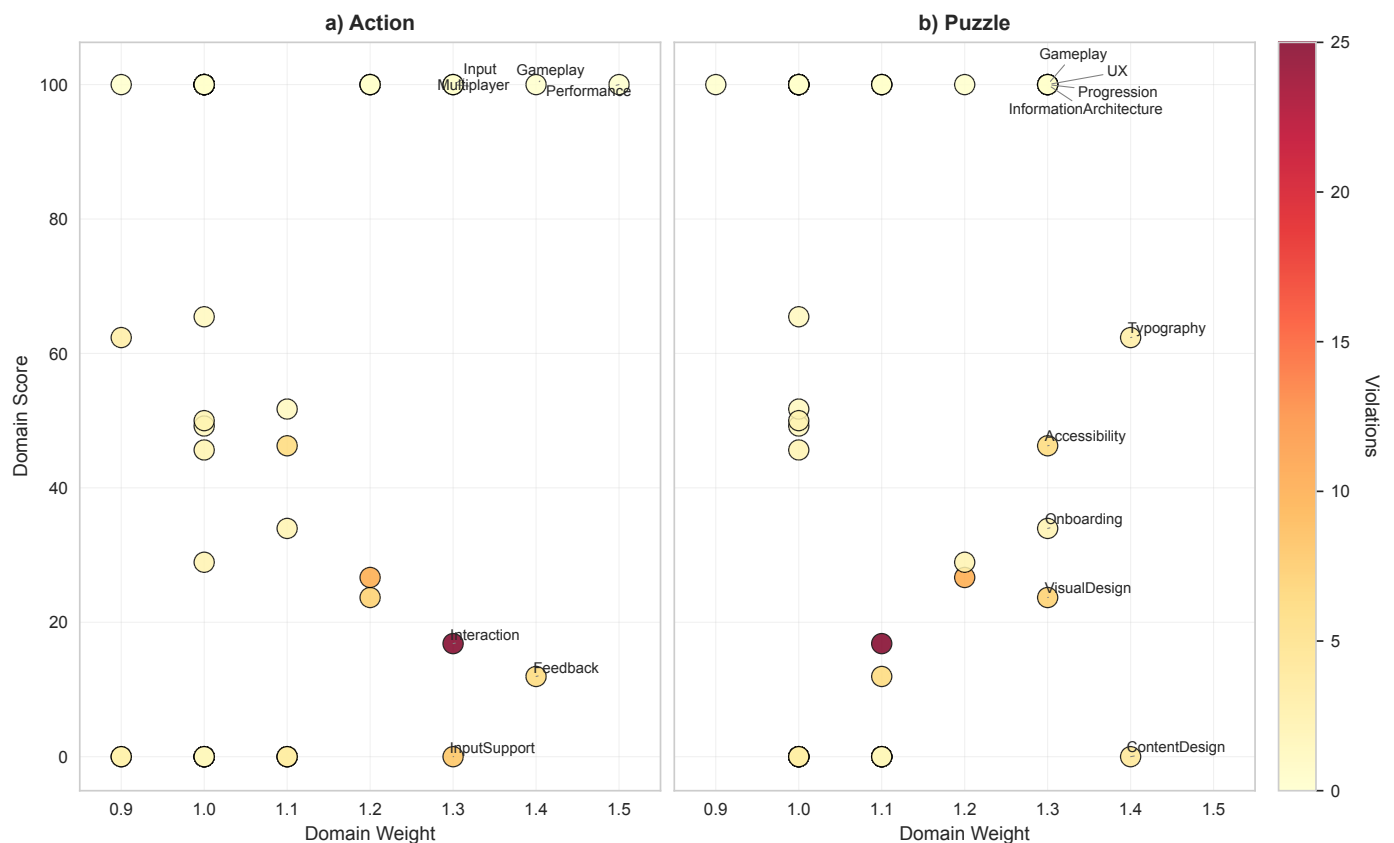


Figure 4. Domain weight versus domain score for Frozen Pengu under two app profiles evaluated with Priority scoring: (a) Action–Priority and (b) Puzzle–Priority. Point color encodes violation count (shared scale, right); priority domains (weight > 1) are labeled. Domain scores are identical between profiles—only the weights change—so each domain holds its vertical position and shifts horizontally between panels, isolating the interpretive role of app profiling. The Action profile amplifies weak interaction-related domains (Interaction, Feedback, Input Support), whereas the Puzzle profile elevates structurally important domains (Content Design, Typography, Visual Design, Accessibility), exposing weak high-priority areas that lower the weighted score.

Table 5. Representative high-severity findings reported for Frozen Pengu.

Rule ID	Domain: Subdomain	Severity	Reported Issue	Recommendation
ACC-005	Accessibility: Hearing Support	High	Media or sound-based feedback may lack captions, transcripts, or non-audio indicators.	Provide captions or transcripts where appropriate and pair critical audio with visual or haptic cues.
ACC-006	Accessibility: Touch Targets and Spacing	High	Interactive elements may be too small or too tightly packed for comfortable use.	Enlarge controls and increase spacing to satisfy platform-recommended target sizes.
BTN-001	Interaction: Hit Target Size	High	Button size or hit region may be smaller than recommended minimums.	Ensure buttons meet minimum tappable-area guidance using padding or frame adjustments.
D4G-004	Interaction: Control Sizing	High	In-game buttons may be visually or physically smaller than recommended platform sizes.	Increase control frames or hit areas and maintain adequate spacing between controls.
D4G-007	Input Support: Platform Defaults	High	The game may rely primarily on a single interaction method rather than platform-default inputs.	Support each platform's expected default input methods in addition to optional controllers.
ACC-009	Accessibility: Timing and Media Control	High	Interfaces may dismiss automatically or media may autoplay without accessible controls or respect for system settings.	Replace auto-dismiss timers with explicit user actions and provide clear, accessible media controls.

6. Discussion

6.1. Findings and Implications

A central implication of this study is that automated UX evaluation should be treated as an executability-aware activity rather than as a binary automation problem. The results indicate that some forms of UX guidance can be checked directly, some can only be approximated through static proxies, and others remain dependent on human review. This is not merely a practical limitation of the current implementation; it reflects a more general property of UX knowledge, which often combines structural, visual, contextual, and experiential considerations. For that reason, an effective automated evaluator should not aim to force all guidance into uniform rule execution, but should instead preserve these distinctions explicitly in its analysis and reporting.

The evaluation also suggests that conservative detection may be preferable to aggressive detection in this setting. In UX-oriented analysis, false positives are especially costly because many findings already require interpretation and prioritization by developers or reviewers [45]. Therefore, a system that produces fewer but more credible findings is often more useful in practice than one that increases coverage at the expense of trust.

At the same time, moderate recall (61.6%) and the corresponding miss rate (38.4%) indicate that current operationalization leaves a substantial portion of UX-relevant conditions undetected even under controlled injection. This gap reflects both the structural sparsity of the evaluation corpus—where 56.1% of planned injections could not be instantiated—and the limitations of proxy-based partial rules. The present framework is therefore better positioned as a triage mechanism that surfaces high-confidence findings and directs reviewer attention than as a comprehensive UX auditor or a substitute for expert judgment.

Another implication concerns the uneven detectability of UX concerns across domains. The variation across categories suggests that static evidence is naturally more informative for some kinds of guidance than for others. This means that progress in automated UX evaluation is unlikely to come from simply adding more rules of the same kind. Instead, different UX domains will likely require different forms of evidence and different operational strategies. Rules related to platform conventions, explicit API usage, or missing support paths may continue to benefit from static analysis, whereas adaptation, presentation quality, and interaction appropriateness may require richer evidence sources.

The project-level reports also point to a practical role for this kind of tool in software development workflows. The value of the system lies less in a single aggregate score than

in its ability to organize findings into interpretable views, such as domain weaknesses, severity groupings, evidence-linked issues, and manual-review recommendations. In that sense, the reporting layer is not a secondary output but a necessary part of the contribution.

It is also instructive to situate TRACEUX explicitly against the most closely related prior work. Mathur et al. [11] proposed a usability evaluation framework for mobile applications based on code analysis, establishing an important precedent that usability knowledge can be at least partially operationalized as code-analysis logic. TRACEUX shares this foundational premise, but differs in scope and architectural design across several dimensions. In terms of scope, Mathur et al. [11] target a defined set of functional usability conditions, whereas TRACEUX operationalizes a broader body of platform UX guidance spanning accessibility, interaction, layout, semantics, content, privacy, and system integration.

Architecturally, TRACEUX introduces three components not present in prior work: (i) an executability-aware rule model that explicitly classifies rules as fully executable, partially executable, or non-executable, preventing overclaiming automation capability; (ii) a context-sensitive weighting model that adjusts domain contributions through application-profile multipliers, enabling different evaluative emphases for different application types; and (iii) a structured recommendation model that links every finding to an actionable remediation directive, transforming the evaluation from a detection system into a decision-support tool. These differences are architectural rather than merely additive: they address the challenge of heterogeneous UX knowledge, where not all guidance can be uniformly operationalized, which is the central design problem that TRACEUX was built to resolve.

A key practical implication of the proposed framework lies in its potential value for App Store submission readiness. Because Apple applications undergo expert review before publication, developers must satisfy not only technical correctness but also broader expectations related to interaction quality, accessibility, platform conventions, and overall UX [46]. In this setting, the formalized Apple HIG rule base and the broader TRACEUX framework could function as an explainable pre-review decision-support layer. Rather than waiting for problems to surface during external review, development teams could use the framework to assess code-level indicators of UX nonconformance earlier, prioritize high-impact issues through context-aware scoring, and act on structured recommendations before submission. As such, the framework has practical relevance not only for academic UX evaluation, but also for improving industrial development workflows in ecosystems where expert platform review plays a gatekeeping role.

Although the corpus is modest in size, it provides meaningful variation along several dimensions relevant to the PoC evaluation objectives. Game type spans three categories (action/arcade, puzzle, and general), ensuring that profile-aware scoring is exercised across structurally distinct configurations. Violation counts range from 78 to 121 across the 12 projects, and overall scores range from 41.6 to 88.0, demonstrating that the instantiated rule base produces differentiated rather than uniformly clustered outputs. Applied injection counts in the controlled validation layer range from 3 (*Rock Paper Scissors*) to 32 (*Potato Game*), further confirming structural heterogeneity within the corpus. These properties are sufficient to support the feasibility and score differentiation claims of our PoC study, while the limitations arising from corpus size and open-source project scope are acknowledged in Section 6.3.

More broadly, the study indicates that automated UX evaluation is most credible when framed as decision support rather than a definitive judgment. The reported outputs can help surface probable problems, reveal recurring design risks, and structure review effort, but they do not directly measure lived UX.

6.2. Limitations and Future Work

The study has several limitations that constrain the strength of its claims. The evaluation is based on a relatively small corpus and a single implementation context, which limits the extent to which the observed behavior can be generalized. In addition, the validation strategy relies on synthetic injection, which is valuable for controlled measurement but cannot fully capture the ambiguity and complexity of naturally occurring UX problems. Therefore, the present findings should be understood as evidence of feasibility and useful behavior under controlled conditions, not as proof of complete real-world coverage.

A further limitation is that the current evidence model is restricted to a static program structure. Many UX properties depend on runtime behavior, visual presentation, temporal dynamics, device state, and user context, all of which are only partially visible or entirely invisible in AST-level analysis. As a result, some rules are operationalized only through coarse proxies, while others cannot be executed at all. This limitation is fundamental enough that future improvements will likely require multimodal evaluation rather than incremental refinement of static rules alone.

Therefore, future work should proceed in at least four directions. First, the evidence base can be extended by integrating artifacts such as rendered interfaces, runtime traces, view hierarchies, interaction logs, or accessibility metadata. Second, empirical evaluation can be strengthened through expert judgment studies and comparisons with human reviewers, which would make it possible to examine not only detection against injected ground truth but also practical usefulness and agreement with expert assessment.

Third, the framework should be tested across broader project types, platforms, and guideline sources, including commercial-grade codebases, to determine which aspects generalize and which require platform-specific adaptation; the present 12-project open-source corpus is not representative of the full diversity of iOS game projects. Fourth, the domain multipliers used in this study were manually specified based on domain knowledge and game UX literature; future work should involve expert elicitation or statistical optimization to validate and refine these weights in a more systematic and reproducible manner.

Another promising direction is methodological rather than purely technical. Instead of treating all UX domains as if they should yield the same kind of evidence, future work can adopt domain-sensitive operationalization strategies. This may include different matching logic, different confidence treatments, or different review pathways depending on whether a rule concerns accessibility, layout, visual hierarchy, platform adaptation, or interaction behavior. Such an approach would better reflect the heterogeneous nature of UX knowledge and could improve both recall and interpretability.

6.3. Threats to Validity

Several threats to validity should be considered.

Construct validity. Formalizing guideline text into static rules inevitably involves interpretive decisions. Different researchers might formalize some rules differently. To mitigate this, the rule schema preserved rationale, severity, confidence, applicability, and reporting metadata.

Internal validity. Synthetic injection tests detection under controlled conditions, but injected violations may not always match the complexity of naturally occurring developer mistakes. The validation therefore demonstrates detection capability under controlled perturbation, not complete ecological realism.

External validity. The framework was evaluated on 12 public Swift game projects. Although this provides useful variation, the sample does not represent all iOS games,

commercial-quality codebases, or non-game apps. Broader studies would strengthen generalizability.

Conclusion validity. The strong precision and moderate recall should not be overgeneralized to all rule families or to all future rule bases. The current numbers are specific to the implemented rules, the selected projects, and the injection strategy. Future expansions of the rule base may change these values.

7. Conclusions

This paper presented TRACEUX, a framework for operationalizing UX guidance as an executable evaluation pipeline, and demonstrated its use through TRACEHIG, a static-analysis-based instantiation for Swift applications. The study showed that UX guidance can be translated into structured rules, evidence queries, scoring logic, and report generation, while also making explicit that different rules support different levels of automation. The empirical evaluation showed that the approach is operationally viable and useful for structured UX inspection.

While the results support the claim that automated evaluation provides meaningful decision support, the study also highlights the boundaries of this static approach. The reported scores reflect rule conformance under a static evidence model—serving as decision-support inputs rather than ground-truth UX quality ratings—as static analysis cannot capture UX in its entirety. Consequently, the core contribution is a principled foundation for combining executable checks, proxy-based inference, and structured manual review. Future work can build on this foundation by incorporating richer evidence sources, expanding cross-platform coverage, and strengthening human-grounded validation.

Author Contributions: All authors have read and agreed to the published version of the manuscript.

Funding: The project was funded by KAU Endowment (WAQF) at King Abdulaziz University, Jeddah, Saudi Arabia. The authors, therefore, acknowledge with thanks WAQF and the Deanship of Scientific Research (DSR) for technical and financial support.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data and artifacts used in this study are available from the corresponding author upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lu, G.; Qu, S.; Chen, Y. Understanding user experience for mobile applications: a systematic literature review. *Discov. Appl. Sci.* **2025**, *7*, 587. <https://doi.org/10.1007/s42452-025-07170-3>.
2. Yıldız Çetin, K. Usability of Mobile Applications: A Conceptualization and Instrument Development Study Based on Apple Human Interface Guidelines. Master's Thesis, Istanbul Technical University: Istanbul, Türkiye, 2023.
3. Nakamura, W.T.; de Oliveira, E.C.; de Oliveira, E.H.; Redmiles, D.; Conte, T. What factors affect the UX in mobile apps? A systematic mapping study on the analysis of app store reviews. *J. Syst. Softw.* **2022**, *193*, 111462. <https://doi.org/10.1016/j.jss.2022.111462>.
4. Muhanna, M.; Masoud, A.; Qusef, A. Usability Heuristics for Evaluating Arabic Mobile Games. *Int. J. Comput. Games Technol.* **2022**, *2022*, 5641486. <https://doi.org/https://doi.org/10.1155/2022/5641486>.
5. Carrión-Toro, M.; Morales-Martínez, D.; Santórum, M.; Bernal, L.; Galindo, J.; Aguilar, J.; Acosta-Vargas, P. Rethinking Usability in Serious Games: Designing an Instrument That Evaluates What Really Matters in Learning Contexts. *Appl. Sci.* **2025**, *15*, 10084. <https://doi.org/10.3390/app151810084>.

6. Umar, M.; Hussain, I.; Mahmood, T.; Mirza, H.T.; Faisal, C.M.N. Design Strategies to Minimize Mobile Usability Issues in Navigation Design Patterns. *Information* **2024**, *15*, 732. <https://doi.org/10.3390/info15110732>.
7. Charoenwet, W.; Thongtanunam, P.; Pham, V.T.; Treude, C. An Empirical Study of Static Analysis Tools for Secure Code Review. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, 16–20 September 2024; ISSTA 2024, p. 691–703. <https://doi.org/10.1145/3650212.3680313>.
8. Gomes, D.; Felix, E.; Aires, F.; Vieira, M. Static Code Analysis for IoT Security: A Systematic Literature Review. *ACM Comput. Surv.* **2025**, *58*, 1–47. <https://doi.org/10.1145/3745019>.
9. Vassallo, C.; Panichella, S.; Proksch, S.; Gall, H.C.; Zeller, A. How Developers Engage with Static Analysis Tools in Different Contexts. *Empir. Softw. Eng.* **2020**, *25*, 1419–1457. <https://doi.org/10.1007/s10664-019-09750-5>.
10. Savanna, L.; Pecorelli, F.; Palomba, F.; De Lucia, A.; Lenarduzzi, V. A Preliminary Study on the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*; Association for Computing Machinery: New York, NY, USA, 2020. <https://doi.org/10.1145/3416505.3423559>.
11. Mathur, N.; Karre, S.A.; Reddy, Y.R. Usability Evaluation Framework for Mobile Apps using Code Analysis. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*; Association for Computing Machinery: New York, NY, USA, 2018; EASE '18, pp. 187–192. <https://doi.org/10.1145/3210459.3210480>.
12. Kamal, O.M.; Nur Shaful, S.M.; Shaziti, A. Heuristic Evaluation: Comparing Generic and Specific Usability Heuristics for Identification of Usability Problems in a Living Museum Mobile Guide App. *Mob. Inf. Syst.* **2018**, *2018*, 1518682. <https://doi.org/10.1155/2018/1518682>.
13. Galavi, Z.; Norouzi, S.; Khajouei, R. Heuristics used for evaluating the usability of mobile health applications: a systematic literature review. *Digit. Health* **2024**, *10*, 20552076241253539. <https://doi.org/10.1177/20552076241253539>.
14. Paz, F.; Huaracha, A.; Paz-Sifuentes, F.A.; Moquillaza, A.; Lecaros, A.; Falconi, F.; Aguirre, J.; Tapia, A. Comparing Two Sets of Usability Heuristics in a User Experience Evaluation: A Case Study in an E-Commerce Mobile Application. In *Proceedings of the Design, User Experience, and Usability*; Marcus, A.; Rosenzweig, E.; Soares, M.M., Eds.; Springer: Cham, Switzerland, 2023; pp. 69–81. https://doi.org/10.1007/978-3-031-35702-2_5.
15. Nugroho, A.; Santosa, P.I.; Hartanto, R. Usability Evaluation Methods of Mobile Applications: A Systematic Literature Review. In Proceedings of the 2022 International Symposium on Information Technology and Digital Innovation (ISITDI), Padang, Indonesia, 27–28 July 2022; pp. 92–95. <https://doi.org/10.1109/ISITDI55734.2022.9944401>.
16. Nasr, E.; Alsaggaf, W.; Sinnari, D. Developing Usability Guidelines for mHealth Applications (UGmHA). *Multimodal Technol. Interact.* **2023**, *7*, 26. <https://doi.org/10.3390/mti7030026>.
17. Sungkaew, K.; Kaewbamrung, C.; Phirom, T.; Chitjaruenyoo, W.; Phopphet, C.; Phrommueang, P.; Yingmuang, S.; Nizam, D.N.M. Playability heuristics for game interface design for children: K. Sungkaew et al. *CCF Trans. Pervasive Comp. Interact.* **2025**, *7*, 30–47.
18. Fang, T. Research on the Integration of Intelligent Human-Computer Interaction in Educational Game Design. In Proceedings of the 2025 International Conference on Electrical Drives, Power Electronics & Engineering (EDPEE), Los Alamitos, CA, USA, 26–28 March 2025; pp. 356–362. <https://doi.org/10.1109/EDPEE65754.2025.00067>.
19. Badkook, N.; Sinnari, D.; Almakky, A. Developing Comprehensive e-Game Design Guidelines to Support Children with Language Delay: A Step-by-Step Approach with Initial Validation. *Multimodal Technol. Interact.* **2025**, *9*, 68. <https://doi.org/10.3390/mti9070068>.
20. Yasin, S.N.S.; Ibrahim, R. Exploring User Experience Evaluation in Mobile Educational Games: A Systematic Literature Review. In Proceedings of the 2024 IEEE 14th Symposium on Computer Applications & Industrial Electronics (ISCAIE), Penang, Malaysia, 24–25 May 2024; pp. 540–545. <https://doi.org/10.1109/ISCAIE61308.2024.10576446>.
21. Feiz, S.; Wu, J.; Zhang, X.; Swearngin, A.; Barik, T.; Nichols, J. Understanding Screen Relationships from Screenshots of Smartphone Applications. In *Proceedings of the 2022 ACM International Conference on Intelligent User Interfaces*; Association for Computing Machinery: New York, NY, USA, 2022. <https://doi.org/10.1145/3490099.3511109>.
22. Baughan, A.; August, T.; Yamashita, N.; Reinecke, K. Keep it Simple: How Visual Complexity and Preferences Impact Search Efficiency on Websites. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*; Association for Computing Machinery: New York, NY, USA, 2020; CHI'20, pp. 1–10. <https://doi.org/10.1145/3313831.3376849>.
23. Zhang, J.; Zhou, J.; Hua, J.; Niu, N.; Liu, C. Mining user privacy concern topics from app reviews. *J. Syst. Softw.* **2025**, *222*, 112355. <https://doi.org/10.1016/j.jss.2025.112355>.
24. Fawad, M.; Rasool, G.; Palma, F. Android Source Code Smells: A Systematic Literature Review. *Softw. Pract. Exp.* **2025**, *55*, 847–882. <https://doi.org/10.1002/spe.3401>.
25. Senanayake, J.; Kalutarage, H.; Al-Kadri, M.O.; Petrovski, A.; Piras, L. Android Source Code Vulnerability Detection: A Systematic Literature Review. *ACM Comput. Surv.* **2023**, *55*, 1–37. <https://doi.org/10.1145/3556974>.
26. Apple Developer. Human Interface Guidelines. 2026. Available online: <https://developer.apple.com/design/human-interface-guidelines> (accessed on 1 January 2026).

27. Domínguez-Álvarez, D.; Gorla, A.; Caballero, J. On the Usage of Programming Languages in the iOS Ecosystem. In Proceedings of the 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), Limassol, Cyprus, 3 October 2022; pp. 176–180. <https://doi.org/10.1109/SCAM55253.2022.00026>.
28. Tkachuk, A.; Bulakh, B. Research of possibilities of default refactoring actions in Swift language. *Technol. Audit Prod. Reserv.* **2022**, *5*, 6–10. <https://doi.org/10.15587/2706-5448.2022.266061>.
29. Laine, T.H.; Lindberg, R.S.N. Designing Engaging Games for Education: A Systematic Literature Review on Game Motivators and Design Principles. *IEEE Trans. Learn. Technol.* **2020**, *13*, 804–821. <https://doi.org/10.1109/TLT.2020.3018503>.
30. Schaffer, N. Heuristic UX Evaluation of Games. In *Game Usability*; CRC Press: Boca Raton, FL, USA, 2022; pp. 105–114.
31. Alzhrani, F. TraceHIG: An Explainable AST-Based and Context-Aware Static UX Evaluator for iOS Apps. 2026. Available online: <https://doi.org/10.5281/zenodo.20362360> (accessed on 23 May 2026).
32. Alzhrani, F. SyntaxScanner: Abstract Syntax Tree Builder. 2026. Available online: <https://github.com/alzhraniFouz/SyntaxScanner> (accessed on 24 May 2026).
33. Scalzo, G. Flappy Swift. 2025. <https://github.com/gscalzo/FlappySwift> (accessed on 1 April 2026).
34. Jimenez, C. Snake Classic. 2023. Available online: <https://github.com/PiXeL16/SnakeClassic> (accessed on 1 April 2026).
35. Schroeder, M. Potato Game. 2025. Available online: <https://github.com/mick-schroeder/PotatoGame> (accessed on 1 April 2026).
36. Kudryashov, V. HyperSnake — iOS Edition. 2025. https://github.com/kv34dev/HyperSnake_iOS (accessed on 1 April 2026).
37. Soria, P.H. Swift4, iOS FlappyBird Clone App. 2021. Available online: <https://github.com/pH-7/Flappy-Canary> (accessed on 1 April 2026).
38. Eleev, A. Physics Vehicle. 2022. Available online: <https://github.com/eleev/physics-vehicle> (accessed on 1 April 2026).
39. Chan, A. Hoop. 2023. Available online: <https://github.com/dscyrescott/Hoop> (accessed on 1 April 2026).
40. Zheng, A. Swift 2048. 2023. Available online: <https://github.com/austinzheng/swift-2048> (accessed on 1 April 2026).
41. Ozel, H. Rock Paper Scissors Game. 2023. Available online: <https://github.com/haliloze1903/StonePaperScissorsGame> (accessed on 1 April 2026).
42. Zhao, B. Spring Festival Crush—Match 3. 2025. Available online: <https://github.com/banghuazhao/Spring-Festival-Crush> (accessed on 1 April 2026).
43. Cooltide. Vintage Frozen Bubble for iOS Re-Implementation. 2023. Available online: <https://github.com/ulritter/FrozenPengu> (accessed on 1 April 2026).
44. Turner, P. Viper Memory Game. 2026. <https://github.com/philipturner/ar-multipendulum> (accessed on 1 April 2026).
45. Lafreniere, B.; R. Jonker, T.; Santosa, S.; Parent, M.; Glueck, M.; Grossman, T.; Benko, H.; Wigdor, D. False Positives vs. False Negatives: The Effects of Recovery Time and Cognitive Costs on Input Error Preference. In *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology*; Association for Computing Machinery: New York, NY, USA, 2021; UIST '21, pp. 54–68. <https://doi.org/10.1145/3472749.3474735>.
46. Apple Developer. App Review. 2026. Available online: <https://developer.apple.com/distribute/app-review> (accessed on 1 January 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.