Journal of the Brazilian Computer Society, 2026, 32:1, doi: 10.5753/jbcs.2026.8239

Decoupling XR Business Logic from Game Engines: An Event-Driven Architecture for VR Safety Training

Abner Augusto Ramos Macedo Antunes de Souza [**Universidade Federal do Ceará** | ***abner.souza@alu.ufc.br***]
José de Paula Barros Neto [**Universidade Federal do Ceará** | ***jpbarros@ufc.br***]
Creto Augusto Vidal [**Universidade Federal do Ceará** | ***cvidal@dc.ufc.br***]
Joaquim Bento Cavalcante Neto [**Universidade Federal do Ceará** | ***joaquimb@dc.ufc.br***]

Universidade Federal do Ceará, Campus do Pici, Bloco 910, 60440-900, Fortaleza – CE, Brazil.

Received: 17 May 2026 • **Accepted:** 01 August 2026 • **Published:** 18 August 2026

Abstract. Virtual Reality (VR) safety training is usually evaluated as a learning intervention; the software architecture that delivers it is seldom the object of study. This paper takes that architecture as its design artefact: an event-driven organisation in which typed events and shared data contracts provide the primary coordination channel between layers, so that training rules, scoring, logging, and scenario data execute independently of the game engine. The design combines a typed `EventBus`, engine-independent C# domain cores, a JSON scenario model, and host-specific adapters, and was instantiated in Unity with the Meta XR SDK for a construction-safety case study covering the Brazilian norms NR-18 and NR-35. Evaluation relies on shared source compiled for three hosts that load one canonical scenario document: the Unity prototype, a Command Line Interface (CLI) harness, and a desktop authoring tool. The shared library, `SafetyProto.Shared`, passes 46 headless tests with 70.3% line coverage, and adding scenarios required no C# edits. Engine and SDK concerns can be separated from domain behaviour without giving up an executable immersive prototype.

Keywords: Virtual Reality, Event-Driven Architecture, Occupational Safety Training, Software Architecture, Construction Safety, NR-18, NR-35, Unity, Engine Independence

1 Introduction

1.1 Context

Occupational accidents in the Brazilian construction industry are a persistent public health burden. The building construction subsector (CNAE 4120 — Classificação Nacional de Atividades Econômicas, the Brazilian national economic activity classification) accounts for 39.9% of work-related disability claims, making it the most affected segment [Ministério Público do Trabalho and Organização Internacional do Trabalho, 2024]. Falls from height are the leading cause of workplace accidents in this subsector, representing 38.8% of accident notifications. Traumatic fractures comprise 67% of external-cause injuries, reflecting the high-energy mechanics of fall and burial events [Ministério Público do Trabalho and Organização Internacional do Trabalho, 2024; Moreira *et al*., 2024]. While regulatory standards NR-18 [Ministério do Trabalho e Emprego, 2022] (construction site conditions) and NR-35 [Ministério do Trabalho e Emprego, 2021] (work at height) mandate hazard training, compliance and effectiveness vary widely.

Immersive technologies — Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR) — increasingly support occupational safety training [Akindele *et al*., 2024; Scorgie *et al*., 2024; Xu and Zheng, 2021]. By safely simulating hazardous conditions, XR systems allow learners to practise decisions, identify risks, and internalise procedures without exposure to real danger. Standalone headsets, hand tracking, and realistic physics simulations accelerate this adoption.

However, VR safety training literature focuses almost exclusively on *pedagogical outcomes*: measuring retention, engagement, presence, and usability via instruments like the System Usability Scale (SUS) [Brooke, 1996], Presence Questionnaire (PQ) [Witmer and Singer, 1998], and Simulator Sickness Questionnaire (SSQ) [Kennedy *et al*., 1993]. The underlying *software architecture* — module structure, communication, maintenance, and extensibility — lacks systematic treatment. In XR systems, rapid iteration across devices, frequent Software Development Kit (SDK) updates, and requirements for reproducible evaluation compound traditional software challenges; design choices in this domain directly determine extensibility, testability, engine-independent validation, and long-term maintainability under evolving XR SDKs.

1.2 Scientific Problem

Immersive training systems built in contemporary game engines typically couple interaction logic, rendering, state management, and rule evaluation within the same component graph. While expedient for prototyping, this monolithic approach hinders long-term maintenance and evolution. Specific problems include:

- **Rigidity:** adding training scenarios or regulatory rules requires modifying co-dependent modules;
- **Device fragility:** tight integration with specific XR SDKs (e.g., Meta XR SDK) propagates SDK updates as breaking changes;

- **Non-testability:** business logic entangled with engine lifecycle methods prevents execution outside the runtime, hindering automated validation;
- **Content authoring bottleneck:** data-driven scenario changes (e.g., new PPE sequences for NR-35) require programmer involvement.

Obukhov *et al*. [2022] report that monolithic virtual training architectures cause high subsystem coupling, making evolution and maintenance costly; they recommend architectural decomposition. Despite this recognition, limited VR safety training research identifies architectural coupling as the root cause of system fragility or proposes principled remedies.

1.3 Gap in the Literature

Literature reviews show limited attention to the *internal architecture* of VR safety training systems. Existing works describe system functions, learning outcomes, and user studies, but omit architectural strategies for maintenance, extension, and validation. While adjacent XR domains apply event-driven design [Ackermann, 2023; Simões *et al*., 2018], occupational safety training lacks this application. The intersection of Event-Driven Architecture (EDA), VR, and Brazilian regulatory compliance (NR-18/NR-35) remains unexplored.

1.4 Objective

This paper proposes and evaluates an **event-driven modular architecture** for decoupling XR business logic from game engines. The occupational-safety application is used as a case study for validating the architecture under concrete regulatory rules and interaction requirements. The thesis is that training rules, scoring, logging, and scenario data can remain executable outside the immersive runtime when communication is mediated by typed events and shared data contracts. We evaluate software engineering properties — modularity, testability, extensibility, and SDK resilience — rather than pedagogical effectiveness. The evaluation uses a three-host implementation: the Unity prototype, an engine-independent CLI harness, and a desktop authoring tool. The hosts compile the same shared domain and scenario-model source and load the canonical JSON scenario through the same loader; its optional scripted playthrough is consumed by the CLI and ignored by Unity.

1.5 Contributions

The main contributions of this work are:

1. A modular Event-Driven Architecture (EDA) for XR safety training, centred on a typed `EventBus`, engine-independent domain cores, and host-specific adapters;
2. A shared JSON scenario model and desktop authoring tool for the three execution hosts;
3. An engine-independent CLI harness and unit-test project providing a replicable validation method for immersive system architectures without requiring the Unity runtime or graphics hardware;
4. An independent evaluation subsystem (`ScoreService` and `SessionLoggerCore`) without Unity runtime dependencies, enabling automated testing in standard CI environments;
5. Empirical evidence from a construction-safety case study showing engine independence, SDK isolation, testability outside Unity, and zero-modification scenario authoring.

The remainder of this paper is organised as follows: Section 2 reviews related work. Section 3 describes the Design Science Research methodology. Section 4 details the architecture. Section 5 reports evaluation results. Section 6 discusses findings, and Section 7 concludes.

2 Related Work

2.1 VR/AR/MR in Construction Safety Training

The pedagogical effectiveness of VR for construction safety training is well established. Meta-analyses by Scorgie *et al*. [2024] and Man *et al*. [2024], together with the state-of-the-art review by Akindele *et al*. [2024], consistently find that VR outperforms traditional instruction in knowledge acquisition, retention, and worker behaviour. Controlled experimental evidence [Nykänen *et al*., 2020; Guo *et al*., 2024; Alzarrad *et al*., 2024] corroborates these findings across multiple cohorts and hazard categories, including the falls-from-height scenarios targeted by NR-35. Specific construction hazards — falls from height, PPE non-compliance, and scaffolding safety — are the most frequently studied [Xu and Zheng, 2021; Joshi *et al*., 2021; Abotaleb *et al*., 2022; Isingizwe *et al*., 2024].

However, this body of work consistently omits discussion of *how* the systems are internally structured. System descriptions provide only functional overviews: which scenarios exist, which assets were used, which hardware was targeted. No paper in this corpus reviews or proposes a principled software architecture for VR safety training as its primary contribution.

2.2 Software Architecture for Immersive Systems

Architectural concerns are almost entirely absent from the VR training literature. When system descriptions appear, they generally report a tightly coupled component structure typical of rapid game-engine prototyping, ignoring the maintenance or extension implications of such design choices.

Obukhov *et al*. [2022] identified this problem directly in virtual training complexes, noting that monolithic architectures create high coupling between subsystems — particularly between simulation, rendering, and evaluation logic — making evolution and maintenance costly; they propose microservice decomposition as the architectural remedy. In the broader game-engine literature, Quamara *et al*. [2024] propose a modular architecture for eXtended Reality systems that separates input handling, scene management, and application logic into independently replaceable components, motivated by the heterogeneity of XR hardware and the frequency of SDK-level changes.

In the broader software engineering literature, architectural patterns such as Model-View-Controller (MVC), Component-Entity-System (ECS), and Event-Driven Architecture (EDA) [Hohpe and Woolf, 2003; Eugster *et al*., 2003] are well-studied, but their application to the specific constraints of immersive training systems — frequent SDK churn, heterogeneous input devices, regulatory compliance requirements — remains systematically underexplored.

2.3 Event-Driven Design in XR Systems

A small but emerging set of works applies event-driven or reactive patterns to XR application development. Ackermann [2023] introduce Event-Condition-Action (ECA) rules as a unified abstraction for AR reactive behaviour, formalising four interaction patterns — instant reaction, timed reaction, continuous evaluation, and chain reaction — that map directly to the typed event payloads used in the architecture presented here.

At the infrastructure level, Simões *et al*. [2018] define an explicit Event-Driven Architecture for an XRM middleware targeting Industry 4.0 processes, implementing mechanisms for event production, detection, consumption, and reaction using a JSON schema for data consistency and connecting industrial processes to digital twins. This work provides formal grounding for the use of EDA in technical XR contexts beyond entertainment applications.

Within training-specific systems, Obeid *et al*. [2025] describe a VR nursing training platform in Unity whose event-driven design allows interactions to be dynamically managed across modular clinical procedure modules. López *et al*. [2021] apply an event-driven architecture to a Mixed Reality platform for occupational safety and health learning, using event prioritisation to manage user input responsiveness under real-time constraints. Castet *et al*. [2024] demonstrate Python-to-Unity decoupling via event-driven architecture, where experimental logic is defined through events and callbacks that communicate with the Unity runtime without direct coupling. Martusciello *et al*. [2025] and Gazis and Katsiri [2024] illustrate event-based and middleware-driven approaches to integrating AR, AI, and gamification layers in interactive applications, further demonstrating decoupled event models' applicability beyond monolithic game-engine architectures.

These works collectively suggest that event-driven decoupling confers advantages in XR contexts analogous to those documented in distributed systems: reduced inter-component coupling, improved testability, and resilience to platform-level changes. However, none addresses occupational safety training as the application domain, or discusses architectural integration with Brazilian regulatory norms.

2.4 Identified Gaps

This review identifies four gaps that our work addresses:

1. No reviewed architecture specifically targets **VR occupational safety training** using an event-driven approach;
2. No **documented, replicable architecture** exists for this domain that could serve as a reference model for future systems;
3. The combination of engine-neutral JSON scenario definitions and a typed `EventBus` for scenario authoring and event routing has not been explored in the reviewed literature;
4. No reviewed work discusses architectural integration with **Brazilian regulatory norms (NR-18, NR-35)** as first-class architectural requirements.

3 Methodology

3.1 Research Type

This work adopts Design Science Research (DSR) [Hevner *et al*., 2004; Peffers *et al*., 2007] because the central objective is to design and evaluate a software architecture artifact rather than observe user behaviour. The artifact is the proposed event-driven architecture; the evaluation focuses on software engineering properties — modularity, testability, extensibility, and engine independence — using the two-layer strategy described in Section 3.4.

3.2 Materials

The following materials were used in this research:

- **VR prototype** — developed in Unity 6 LTS (version 6000.3.14f1) with the Meta XR SDK v201.0.0, targeting the Meta Quest 3 headset with hand tracking as the primary input modality;
- **EDA modular architecture** — implemented in C# within the prototype, comprising the components described in Section 4;
- **JSON scenario definitions** — engine-neutral files encoding safety tasks, PPE requirements, scoring weights, and task sequences for the implemented construction site scenario;
- **Unit test suite** — pure C# tests with no Unity dependency, covering the core business logic modules;
- **CLI harness** — a console runner that loads scenario data, instantiates all logic modules, and simulates full training sessions without any graphics engine.

3.3 Procedure

The research followed five steps:

1. **Architectural requirements elicitation** — requirements were derived from the construction safety literature, Brazilian regulatory norms NR-18 and NR-35, and analysis of architectural limitations reported in existing VR training systems;
2. **Architecture design** — the EDA architecture was designed and each layer justified against the elicited requirements (Section 4);
3. **Prototype implementation** — the architecture was implemented in Unity using a typed `EventBus`, engine-neutral scenario data, and logically decoupled gameplay and evaluation logic;
4. **Architecture evaluation** — the architecture was evaluated using the two-layer strategy described in Section 3.4,

producing quantitative evidence of modularity and testability;

5. **Analysis and discussion** — results were interpreted against the software quality criteria and connected to the identified literature gaps.

3.4 Evaluation Strategy

The evaluation uses two complementary layers. Layer 1 validates the correctness of individual modules in isolation; Layer 2 validates decoupling at the system level. This two-layer strategy runs against the three-host implementation introduced in Section 1: the unit suite and CLI harness compile the shared domain source directly, while Unity compiles the same source into its own assemblies; all three hosts use the shared scenario schema and loader. Together they constitute the primary empirical contribution of this work.

Test cases were derived from three sources: the safety constraints encoded from NR-18 and NR-35, the expected event sequence of each scenario step, and failure modes observed during prototype implementation. For each rule, the expected player action was mapped to an event payload, the domain component responsible for interpreting that payload, and the terminal event that should be published. A PPE requirement therefore becomes a positive case (the required equipment is worn and the task completes), a negative case (the wrong equipment triggers a violation or penalty), and an ordering case when the task belongs to a sequential group. Scaffolding-inspection rules follow the same structure: interaction events drive task completion, group dependencies gate access to later work, and timeout or missing-correction paths must publish terminal safety events.

3.4.1 Layer 1 — Unit Test Suite

A pure C# unit test suite exercises each core module independently, with no dependency on the Unity runtime. Fixtures cover `ScoreRuleEngineCore`, `SafetyRuleEngineCore`, `TaskManagerCore`, the `PPEManager` event protocol, `SafetyPatternDetector`, and supporting diagnostics. Tests cover correct scoring for valid and invalid PPE selections, PPE validation against NR-35 requirements, task state transitions, event ordering, and scenario-load validation.

Representative test cases include:

- `HappyPath_FullPpeInspectionScenario_MatchesCliParity` — replays the nine-task script and asserts 9/9 completion, a 1,400-point score, and the ordered session/group lifecycle;
- `PpeViolation_ActionWithoutRequiredPpe_RaisesViolationAppliesPenalty` — asserts violation publication and end-to-end PPE-penalty application;
- `Sequential_OutOfOrderAction_IsRejected` — asserts that an action for a later sequential task cannot complete early;
- `LoadValidation_MalformedJson_ReturnsFailureDoesNotThrow` — asserts controlled rejection of malformed scenario input.

Metrics reported: the full suite comprises 52 test methods — 44 unit tests and 8 integration tests. Of these, 46 execute headlessly through `dotnet test` (all 8 integration tests and 38 of the 44 unit tests); the remaining 6 unit tests require the Unity Editor and are excluded from the headless count and the coverage figure. Beyond verifying correctness, the isolation property proved diagnostically valuable: a logical defect in `SafetyRuleEngineCore` involving ambiguous handler registration was reproduced and localised entirely within the NUnit suite, eliminating a family of Unity-specific hypotheses (Script Execution Order, `ScriptableObject` lifecycle, queue race conditions) in a single test run.

The integration battery is implemented in `SessionIntegrationTests`. It wires the production domain stack through a real `IEventBus` port and checks emergent behaviour across task management, safety-rule evaluation, scoring, and logging. The driver is the scripted test body, executed through `SessionTestHarness.WearPpe`, `Attempt`, and `ReplayScript`; this is the in-process equivalent of the CLI `ScriptedActor`. Its stubs are `FakeEventBus`, which provides deterministic enqueue-then-drain delivery in place of Unity's frame-flushed `EventBus`, and PPE-sensor events standing in for trigger callbacks from `PPEManager` and `PPEZone`. The domain components themselves are not stubs: `TaskManagerCore`, `SafetyRuleEngineCore`, `ScoreRuleEngineCore`, `ScoreService`, and `SessionLoggerCore` run as production code. Eight integration tests cover CLI–domain parity, PPE violation scoring, event ordering, dependency gating, timeout terminality, and load validation for unknown PPE and malformed JSON.

3.4.2 Layer 2 — CLI Harness

A console runner in pure C# loads the canonical scenario document embedded in the Unity project through the same shared loader, instantiates all logic modules, simulates user inputs, and fires events through the `HarnessEventBus`, producing a full session transcript without any graphics engine, headset, or Unity installation. This shows that the business logic layer operates independently of the immersive runtime: when the same modules execute in a console context, the architecture's decoupling is exercised in practice rather than asserted only by source inspection. The CLI harness output is reported as a listing in Section 5 and is fully reproducible from the public repository.

This layer evaluates protocol extensibility rather than source-level portability. The relevant claim is not that every module executes unchanged in both runtimes, since components such as `PPEManager` are legitimately tied to engine-specific physics. Instead, the architecture exposes a well-defined event protocol, centred on `IEventBus` and shared payload contracts, through which new participants such as stubs, external agents, and analysis tools can join the system without modifying existing modules. The CLI harness is therefore a concrete participant in that protocol, not evidence that all source files are portable across runtimes.

4 Proposed Architecture

4.1 Overview

The architecture organises the VR safety-training system into seven layers connected via a typed `EventBus` ScriptableObject acting as the central communication spine. Layers coordinate via typed event payloads without direct references. Following a ports-and-adapters pattern, pure domain logic resides in engine-independent C# core classes (`*Core`), wrapped by Unity `MonoBehaviour` adapters for scene integration. Figure 1 groups the Adapter layer and UI consumers for compactness; the following subsections detail all seven layers.

4.2 Layer 1 — Data Configuration

Scenario content is stored as engine-neutral JSON files, separated from code and engine-specific scene assets. In this architecture, a scenario is a single JSON document specifying the task groups, individual safety tasks, scoring rules, and PPE requirements of one training session; replacing this document changes the training content and task flow without modifying code, scenes, or binaries. The same schema is consumed by the Unity prototype, the CLI harness, and the desktop authoring application through `SafetyProto.Shared`, which compiles the shared scenario model and loader outside Unity. Three definitions compose this layer: `ScenarioDef` identifies the scenario and its top-level metadata; `TaskGroupDef` clusters tasks under an execution mode (sequential or free-order) and may declare dependencies on other groups; and `SafetyTaskDef` describes a trainable action, its scoring rules, and its PPE or inspection requirements. The interaction vocabulary is validated against the shared action-capability catalog, so scene interactables, scenario files, and headless tests refer to the same identifiers. This vocabulary is closed at authoring time: catalog entries, their Unity interactables, and the corresponding PPE models and 3D scenes are produced in advance by developers and artists, and scenario authors compose training sessions from these predefined elements rather than introducing new ones.

At runtime, `ScenarioLoader` first attempts to load an external override from the application's persistent-data directory, which allows a Quest build to receive a revised scenario through `adb push` without rebuilding the application. If no override is present, the loader falls back to the embedded default scenario in `Resources/Scenarios/`. The same schema and loader therefore operate in three contexts without a format translation step: authoring-time editing in the Avalonia tool or a text editor, runtime loading in a host that compiles `SafetyProto.Shared`, and business-logic execution in classes with no engine dependency. At authoring time, scenario composition can be performed through the desktop tool or a text editor without Unity, although creating new 3D assets and interaction capabilities remains Unity-specific developer work. At runtime, Unity-specific publishers and adapters connect the immersive scene to the shared event protocol, whereas the CLI executes without Unity. At the business-logic level, the shared scenario loader and domain components compile and execute in pure .NET. In the implemented codebase, 31 shared-library files compile through the CLI-linked project alongside 16 Domain Core files (2,172 LoC) protected by `noEngineReferences:true`; these figures make the data/runtime/business-logic separation build-visible rather than only conceptual. Table 2 separately reports 43 files with no textual `UnityEngine` reference; this lexical indicator is broader than the build-verified 31-file shared project and is not treated as equivalent evidence.

The desktop authoring tool (`Tools/AuthoringApp.Gui`) provides a non-Unity editing surface for files under the shared scenario schema. It performs two validation passes before a scenario is used: structural validation of the scenario model and semantic validation against the capability catalog. This host-neutral workflow preserves the no-code scenario-authoring requirement without tying content editing to the Unity runtime. Figure 2 shows this editing surface.

4.3 Layer 2 — Unity Publishers

This layer translates physical player interaction and session lifecycle into typed events on the bus. Its components are pure producers: they never subscribe. Its producers fall into three categories. Action producers (`ActionEmitter` and `ActionTrigger`) emit a canonical interaction event whenever the player engages a scene object via grab, proximity, or interaction-SDK callback. A PPE tracker (`PPEManager`) publishes equipment-state changes when the player dons or removes protective items. A session manager (`TrainingSessionManager`) issues session-lifecycle events — started, paused, resumed, ended — in response to engine and application focus callbacks. Every event payload is stamped at dispatch time with a session-context record so that downstream consumers can correlate events to a specific training run.

4.4 Layer 3 — Central Spine

The `EventBus` is the single typed publish/subscribe broker [Eugster *et al*., 2003] through which every cross-layer interaction flows. It exposes generic subscription, unsubscription, and publication primitives parameterised by event payload type, and flushes its queue once per frame to guarantee consistent dispatch order with respect to the engine's update cycle. Two design properties have direct implications for the evaluation. First, the bus stamps every payload with session-context metadata at dispatch time, which gives downstream consumers a uniform correlation key without requiring producers to share state. Second, the bus enforces an ordering invariant between subscribers and producers during session initialisation: consumers must be registered before producers become active. The codebase enforces this ordering in the session lifecycle: `TaskManager` loads the scenario in `Awake`, registers its consumers in `Start`, and activates the first task only after receiving `SessionStarted` from `TrainingSessionManager`.

4.5 Layer 4 — Adapters and Unity Consumers

This layer is the seam between the engine and the engine-independent core. It serves two complementary functions. The first is wrapping: each domain core (Layer 5) has a

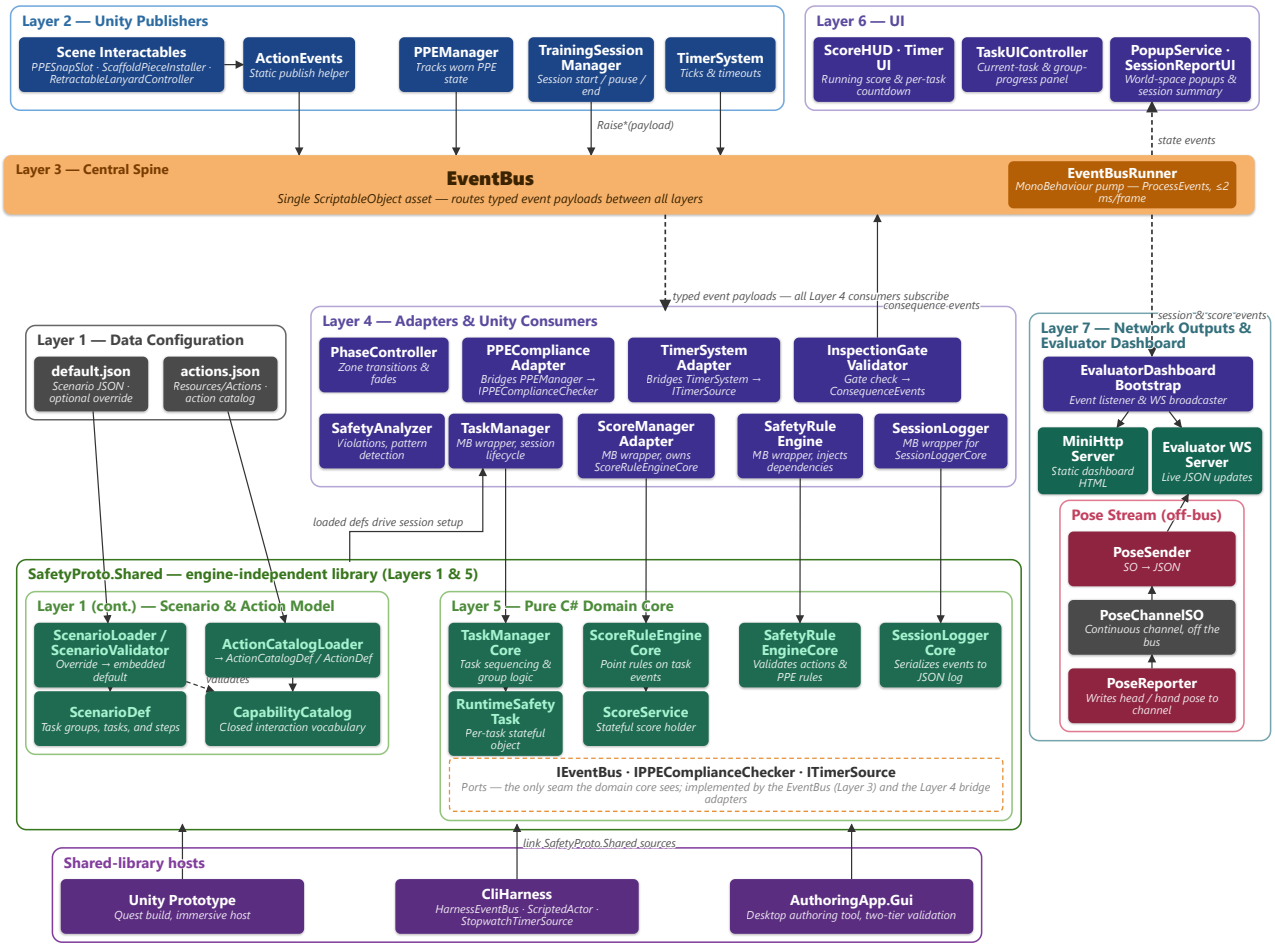

Figure 1. Layered event-driven architecture of the `safety-training-proto` system, with a shared JSON scenario source, engine-independent domain core, and host-specific Unity adapters. MB: MonoBehaviour.

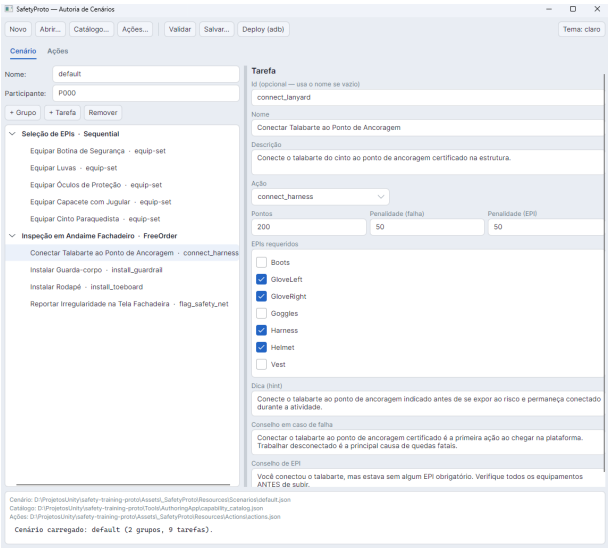

Figure 2. Form-based editing of a scenario in the desktop authoring tool, validated against the scenario model and the action-capability catalog.

thin engine-side wrapper that subscribes to the bus, forwards events into the core, and publishes any resulting events back onto the bus. The second is engine-coupled consumption: components that legitimately require engine services — such as scene transitions and screen fades — listen directly to the bus and act on Unity APIs without traversing the domain core. Two consequences follow from this division. Dependency injection is concentrated here: wrappers are the only place where engine-specific implementations (for example, a Unity-based timer source or PPE compliance checker) are bound to the abstract ports consumed by the core. SDK migration risk is similarly contained: the breaking surface of an engine or XR-SDK update propagates only within Layers 2 and 4, leaving Layers 5–7 untouched (justified empirically in Section 5).

4.6 Layer 5 — Pure C# Domain Core

The domain core is where the primary architectural claims are testable: its classes carry no engine dependencies and are fully exercisable via the CLI harness and the standalone unit-test project.

`TaskManagerCore` owns task and group sequencing. On each interaction event it advances the matching task, emits the corresponding lifecycle event back onto the bus, and handles

group dependencies, timeouts, and order-violation tracking in both sequential and free-order execution modes. Figure 3 illustrates the resulting state machine; the diagram makes explicit that every state transition is published as a typed event, with no direct reference between components.

`SafetyRuleEngineCore` validates each action against the currently active tasks and checks PPE compliance through an abstract port, emitting a typed violation event when a regulatory constraint is breached. `ScoreRuleEngineCore` listens for task-lifecycle events and applies scoring or penalty rules through an abstract score service, keeping all scoring policy inside the core and free of engine dependencies. `ScoreService` is a stateful score holder that exposes mutation and notification primitives but does not itself subscribe to the bus, allowing it to be unit-tested in isolation. `SessionLoggerCore` subscribes to every bus event, appends timestamped entries to an in-memory log, and at session completion serialises that log through a serialisation strategy supplied via dependency injection. The injected-strategy pattern enables the same logger to operate both inside the engine and inside the harness: each host binds its own serialiser without modifying the core.

4.7 Layer 6 — UI

The UI layer does not own authoritative domain state. Its components primarily subscribe to the bus and update presentation, although UI services may publish session-control events such as pause and resume. In-scenario surfaces include a running-score HUD, a per-task timer, a current-task and group-progress panel, a session-completion summary, and a world-space popup service used for both onboarding and in-session warnings. Because the UI consumes the same event protocol as every other layer, alternative presentations — a desktop dashboard, a non-immersive spectator view, a future MR overlay — can be introduced without modifying the producers or the domain core, provided they implement the same subscription contract.

4.8 Layer 7 — Network Outputs and Evaluator Dashboard

This layer enables real-time remote observation by a trainer or researcher over the local network, without any intervention in the running simulation. A bootstrap component starts an HTTP server (for the static dashboard page) and a WebSocket server (for live event streaming) at device startup, subscribes to every event on the bus, and broadcasts the payloads as JSON to connected browsers. The layer also includes a separate channel for high-frequency pose data (head, hands, PPE-object transforms at approximately 10 Hz): to prevent continuous telemetry from congesting the discrete-event channel, pose samples are written to a dedicated `ScriptableObject` channel inside the engine's late-update phase and forwarded through the same WebSocket server on a timed cadence rather than through the bus. This separation between discrete and continuous channels is the implementation counterpart to the frame-budget discussion in Section 4.10.

4.9 Event Catalog

The prototype defines 18 logical events of which 16 are dispatched as typed `EventBus` channels (Table 1); two animation-coupled events use an off-bus synchronous channel. These 16 channels are implemented by 13 payload classes, since related events share a class distinguished by a phase enum (e.g., `TaskEventArgs` carries `Started`/`Completed`/`Timeout`). Modules never inspect raw input or query external state directly. Events fall into six groups: 1) *session lifecycle* (`SessionStarted` to `SessionCompleted`), managed by `TrainingSessionManager` and `TaskManagerCore`; 2) *player interaction* (`ActionAttempt`), fired on user engagement and consumed by PPE, scoring, and logging; 3) *PPE compliance* (`PpeStateChanged`), published by `PPEManager` after NR-35 evaluation; 4) *task and group lifecycle* (`TaskStarted` to `GroupCompleted`), published by `TaskManagerCore` for scoring and UI; 5) *safety analysis* (`SafetyViolation` to `SafetyError`), produced when regulatory constraints are breached; and 6) *consequence system* (`ConsequenceStarted`, `ConsequenceEnded`), triggering corrective feedback. Table 1 lists the events and their primary publishers.

Table 1. Event catalog of the `safety-training-proto` prototype.

Event	Publisher
`SessionStarted`	`TrainingSessionManager`
`SessionPaused`	`TrainingSessionManager`
`SessionResumed`	`TrainingSessionManager`
`SessionEnded`	`TrainingSessionManager`
`SessionCompleted`	`TaskManagerCore`
`ActionAttempt`	`ActionEmitter`/`ActionTrigger`
`PpeStateChanged`	`PPEManager`
`TaskStarted`	`TaskManagerCore`
`TaskCompleted`	`TaskManagerCore`
`TaskTimeout`	`TaskManagerCore`
`GroupStarted`	`TaskManagerCore`
`GroupCompleted`	`TaskManagerCore`
`ScoreChanged`	`ScoreService`
`SafetyViolation`	`SafetyRuleEngineCore`
`CriticalSafetyFailure`	`SafetyAnalyzer`
`SafetyError`	Multiple
`ConsequenceStarted`†	`InspectionGateValidator`
`ConsequenceEnded`†	`InspectionGateValidator`

† Dispatched through an off-bus synchronous channel rather than the typed `EventBus`, because their Unity-coupled payload and animation-timing semantics are incompatible with the bus.

4.10 Architectural Rationale

Five decisions warrant justification.

Typed events over string-keyed messages. Typed event payloads eliminate runtime errors from typos and enable compile-time checks. Migrating from string keys resolved three latent subscriber-mismatch defects identified during integration.

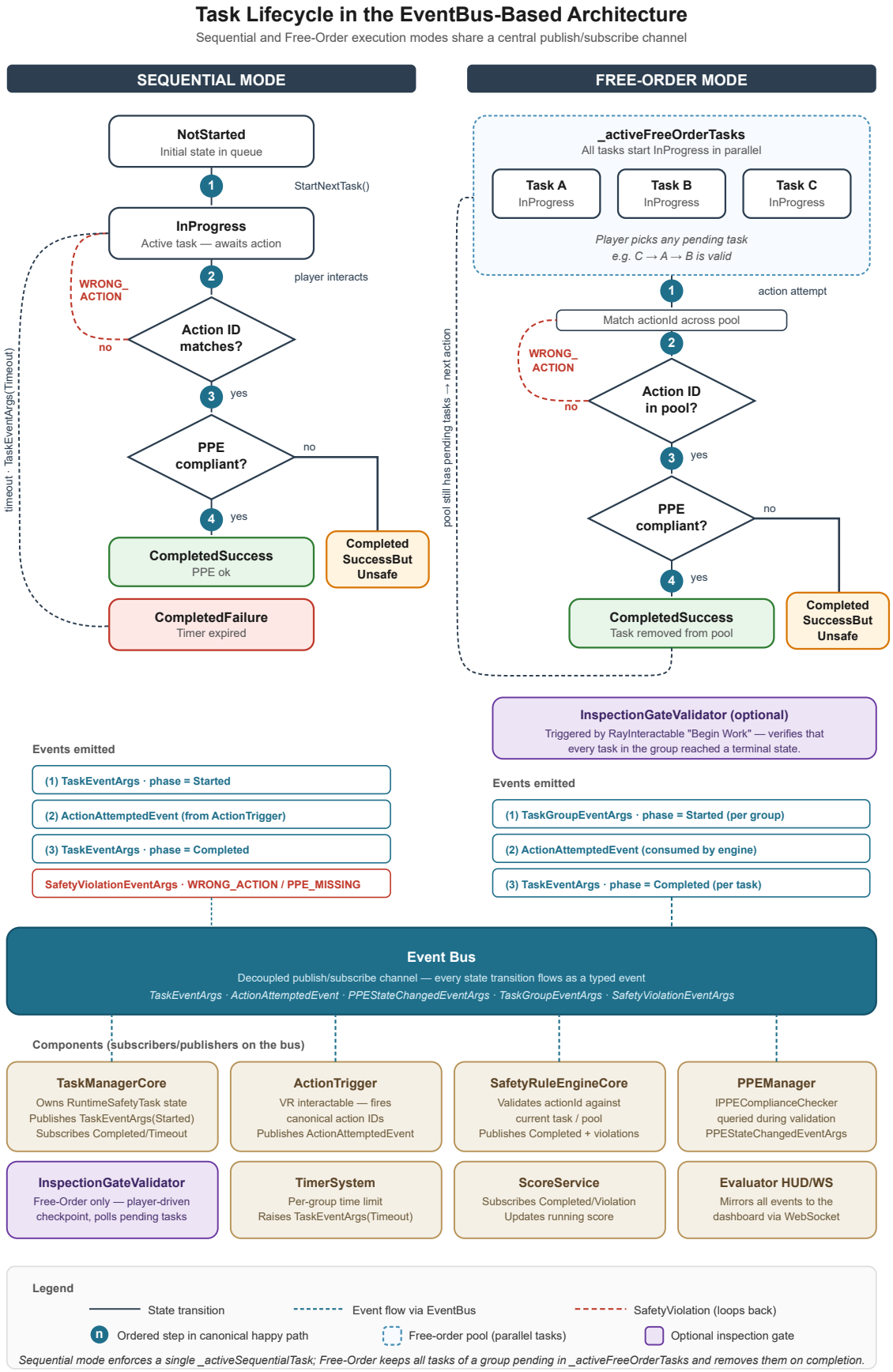

Figure 3. Task lifecycle state machine for `Sequential` and `FreeOrder` execution modes. Numbered circles trace the canonical happy path. Every state transition is published as a typed event on the `EventBus`; no component holds a direct reference to `TaskManagerCore`.

Engine-neutral JSON as the scenario source. The data layer uses one JSON schema and shared loader across the Unity prototype, CLI harness, and desktop authoring tool. This choice eliminates data divergence between hosts: a new host links `SafetyProto.Shared`, implements presentation adapters, and loads the same `ScenarioDef`/`TaskGroupDef`/`SafetyTaskDef` files. The trade-off is that content authors do not rely on Unity Inspector conveniences such as object pickers and inline asset references. The Avalonia authoring tool addresses this by providing form-based editing and two-tier validation, while retaining plain-text diff, branch, and review workflows. XML would provide comparable engine neutrality; JSON was preferred because its object model maps directly onto the scenario definitions and because it is the common interchange format outside the engine. The Unity prototype, CLI harness, integration tests, and desktop authoring tool use the canonical `Resources/Scenarios/default.json`. Its optional `script` field contains test-driver metadata used only to automate CLI execution; it does not alter task, scoring, or safety semantics and is ignored by hosts that do not require scripted execution.

SDK coupling confined to Publisher and Adapter layers. The Domain Core, Evaluation subsystem, and UI layer have zero SDK dependencies. Coupling is confined to the Publisher and Adapter layers (e.g., `OVRScreenFade`, `Oculus.Interaction`), which bridge platform-specific input and business logic. In the tracked Meta XR SDK migration, changes remained in these boundary layers and configuration files; Domain Core, Evaluation, and UI source files were unaffected. This isolation enables substituting XR input with a console reader in the CLI harness while leaving downstream logic intact.

Serialisation injected as a delegate. `SessionLoggerCore` receives its serialisation strategy through dependency injection rather than hardcoding an engine-specific serialiser, allowing context-specific implementations and future format support without modifying core logic.

Frame-budgeted dispatch for standalone XR runtimes. Standalone XR applications operate under tight CPU budgets (8–14 ms for 72–120 Hz). Unbounded synchronous event dispatch can degrade frame pacing during bursts. The `EventBus` addresses this via three parameters: 1) `EventBusRunner` bounds dispatch to 2 ms/frame via its `ProcessEvents` step (≈15–25% of budget), deferring remaining events; 2) a 1000-entry queue limit triggers a `SafetyError`, preventing silent backlog accumulation; and 3) high-frequency pose data (≈10 Hz) bypasses the `EventBus` via `PoseChannelSO`, preventing telemetry from competing with gameplay events. These are design-time mitigations; empirical performance characterisation remains future work (Section 6.1).

The architecture is extensible through its ports: delivery timing is implementation-defined behind `IEventBus`, and the scenario schema is consumable by external tools, including an MCP server for automated scenario-authoring workflows.

5 Case Study

5.1 Objective

This case study instantiates the proposed EDA architecture in a construction-safety training scenario and evaluates it against software engineering criteria. We do not assess pedagogical effectiveness; user studies measuring learning outcomes, presence, or usability are deferred to future work. We evaluate the architectural properties from Section 4: modularity, testability, extensibility, and SDK resilience.

5.2 Implemented Scenario

The prototype implements a construction site safety training scenario addressing two NR-35 hazard categories: PPE selection and scaffolding inspection. The scenario comprises nine tasks in two groups, encoded in a shared JSON scenario file loaded by Unity and the CLI harness.

Group 1: PPE Selection (`ppe_selection`). Five tasks, `Sequential` execution, 420 s limit in the Unity scenario. Trainees must equip: safety boots, gloves, goggles, a helmet with chin strap (NR-35), and a parachute harness. Maximum score: 750 points. The PPE workbench includes correct items and distractors: a helmet without a chin strap, an abdominal belt (prohibited by NR-35 for height work), latex gloves (inadequate for metallic structures), and open-toe footwear (prohibited by NR-18). These distractors force trainees to make selection decisions grounded in NR-35 rather than equipping any item.

Group 2: Scaffolding Inspection (`scaffold_inspection`). Four tasks, `FreeOrder` execution, 300 s limit, with a `requiredGroups` dependency on Group 1. `FreeOrder` mode reflects real-world practice, where workers address irregularities as encountered. Tasks: connect the harness lanyard to an anchor point (type A1 per ABNT NBR 16325 [Associação Brasileira de Normas Técnicas, 2024]), install a missing guardrail (NR-18 §18.15), install a missing toeboard (NR-18 §18.15), and flag a fallen net. Maximum score: 650 points. The authored nine-task scenario totals 1,400 points in the CLI harness.

The scaffolding platform contains four irregularities: a missing guardrail, a missing toeboard, a disconnected lanyard, and a fallen safety net. If the trainee presses "Start Activity" without correcting these, `InspectionGateValidator` triggers consequences — a tool falling from the edge (guardrail) and a simulated fall (disconnected lanyard). Each consequence is logged as a `SafetyViolation` or `CriticalSafetyFailure` with a -100 point penalty.

The virtual environment uses a Unity scene with two zones: `ZonaCanteiro` (PPE selection) and `ZonaAndaime` (scaffolding). Only the active zone renders. Upon Group 1 completion, the `PhaseController` fades to black and teleports the player to `ZonaAndaime`; the event cycle handles dependency resolution and group bootstrapping, producing a `GroupCompleted` → `GroupStarted` transition within the event-dispatch cycle.

In a representative unsafe-action flow, the user attempts to connect the lanyard while required PPE is missing. The interaction adapter publishes `ActionAttemptedEvent`; `SafetyRuleEngineCore` evaluates the action against

the active task and cached PPE state, then publishes `SafetyViolation` and a non-compliant task completion. `ScoreRuleEngineCore` applies the task's PPE penalty through `ScoreService`, while `SessionLoggerCore` and the UI consume the resulting events independently.

Figure 4 shows the two zones of the implemented scenario, and Figure 5 shows the evaluator dashboard through which this event traffic was observed during on-device sessions.

5.3 Evaluation Criteria

We measured the following criteria:

- **Scenario addition effort** — lines of code and modules modified to add a second scenario;
- **Safety rule addition effort** — effort to add a `SafetyTaskDef` with a novel PPE rule without engine recompilation;
- **UI modification independence** — UI feedback layer modification independence from `TaskManager` or `ScoreService`;
- **SDK migration impact** — modules modified during a Meta XR SDK update;
- **Unit test coverage** — passing tests and business logic branches covered without Unity;
- **CLI harness execution** — complete session execution without graphics, headset, or Unity.

5.4 Results

Figure 6 excerpts a CLI harness execution at significant moments; the complete trace can be regenerated with the command in Section 5.5.

The CLI harness ran the two-group scenario, completing all nine scripted tasks and scoring 1,400 points. The same runner completed the five-task PPE-only scenario with a score of 750 points. Both hosts apply the scoring rules through the shared domain core; the reported 1,400-point session total is the CLI result.

CLI execution confirms that task sequencing, score computation, and event dispatch execute outside Unity. The Group 1 to Group 2 transition shows that dependency resolution and group bootstrapping execute within the event dispatch cycle. The harness also supports an interactive mode for manual PPE selection to exercise error paths and penalty logic.

The test suite contains 52 test methods across eight fixtures (44 unit and 8 integration; decomposed in Section 3.4.1). Forty-six tests run headlessly through `dotnet test` without the Unity Editor; the remaining six unit tests require the Unity Editor and are excluded from the coverage figure. Table 2 summarises the execution results and codebase indicators.

The dependency graph supports the engine-independence claim more directly than raw file counts. Table 3 reports afferent coupling (Ca), efferent coupling (Ce), and instability $I = Ce/(Ca + Ce)$ for the own-module reference graph. The depended-upon modules sit at the stable end: `EventBus.Core` has $I = 0.00$, `Domain.Core` has $I = 0.17$, and the .NET mirror `SafetyProto.Shared` has $I = 0.00$. Leaf Unity hosts such as `UI.Unity`, `Networking.Unity`, and `Editor` have $I = 1.00$. No cycles were found, and

Table 2. Software quality evaluation results and codebase indicators.

Evaluation results	
Total test methods	52
Headless tests passed (no Unity Editor)	46 / 46
Integration tests	8
Line coverage, shared library (`SafetyProto.Shared`)	70.3%
Branch coverage, shared library (`SafetyProto.Shared`)	57.3%
CLI scenario: PPE selection	5/5 tasks, 750 points
CLI scenario: PPE inspection	9/9 tasks, 1,400 points
Codebase indicators	
Total source files (excl. tests/editor)	120
Total source LoC	15,170
Files without textual `UnityEngine` reference	43
LoC without textual `UnityEngine` reference	3,305
Domain Core files	16
Domain Core LoC	2,172
Shared-library files (CLI-linked)	31
Harness-specific LoC	804 (8 files)
Typed `EventBus` channels	16
Event payload types	13
Files importing Meta XR SDK	17

`Domain.Core.asmdef` sets `noEngineReferences:true`. Abstractness — the ratio of abstract to concrete types per module — was not computed: the domain layer is organised as concrete classes coordinated through narrow interface ports (`IEventBus`, `IScoreService`) rather than an abstract-class hierarchy, so a type-count ratio would not characterise this design meaningfully; instability and the dependency direction reported above capture the property most relevant to the reviewers' request.

Table 3. Module coupling in the own-module dependency graph.

Module	Ce	Ca	I
`EventBus.Core`	0	6	0.00
`Domain.Core`	1	5	0.17
`Utils.Unity`	2	4	0.33
`Runtime.Unity`	3	3	0.50
`UI.Unity`	4	0	1.00
`Networking.Unity`	4	0	1.00
`Editor`	4	0	1.00
`SafetyProto.Shared`	0	3	0.00

Change-impact measurements show where modifications occurred. Table 4 reports the tracked implementation episodes used as evidence for decoupling: a data-layer introduction, an SDK migration, a cross-cutting feature addition, and scenario authoring.

Table 5 centralises the evidence for the main architectural claims. Each row points to a concrete artifact, test, metric, or

(a) (b)

Figure 4. The two zones of the implemented scenario. (a) The Group 1 PPE workbench, holding the five required items and the distractors (helmet without chin strap, abdominal belt, latex gloves, and open-toe footwear). (b) The Group 2 scaffolding platform, with the four seeded irregularities annotated: missing guardrail, missing toeboard, disconnected lanyard, and fallen safety net.

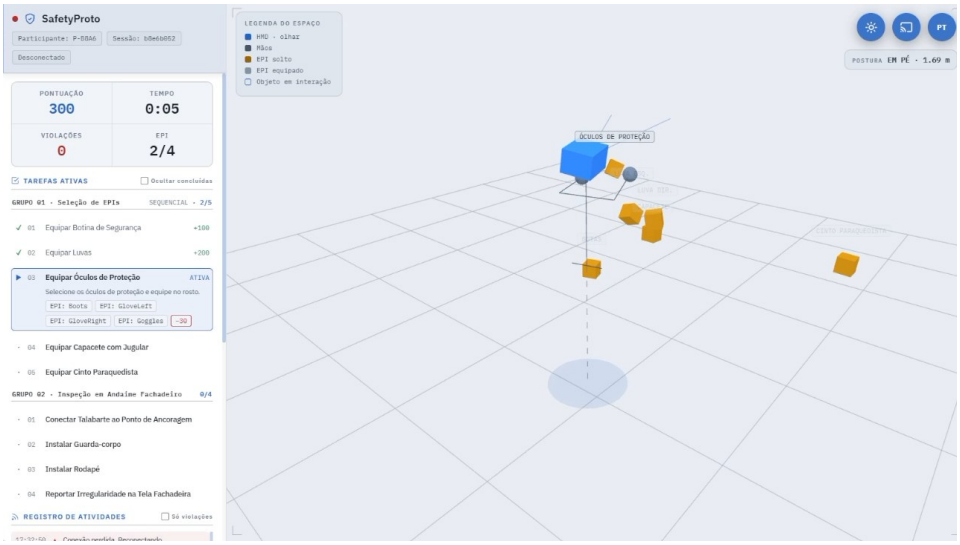

Figure 5. Evaluator dashboard rendered in a web browser on the local network during a session: current score, per-task states, and a live three-dimensional view of the headset, hand, and PPE-object poses streamed at approximately 10 Hz (Section 4.8).

commit-level measurement reported above.

5.5 Reproducibility

Reproducing these results requires only the prototype repository: the shared scenario files, the CLI harness, and the engine-independent test project. The repository contains the Unity prototype, `Tools/SafetyProto.Shared`, `Tools/CliHarness`, `Tools/SafetyProto.Tests`, the scenario files, and the benchmark project used for Table 6. The evaluation snapshot is the repository release `v1.0.0` (commit `d8d16d2`); its attached evidence package contains the runner logs, TRX results, Cobertura report, CLI session logs, and machine-readable summary. Figure 7 lists the commands, all run from the repository root.

The PowerShell and Bash wrapper commands provide equivalent Windows and macOS/Linux workflows. Each validates the expected counts, coverage, and CLI outcomes and writes its logs and summary under `artifacts/reproduction/`. The first individual command is the runner of record for the 46 headless tests. The second isolates the eight integration tests described in Section 3.4.1. The third command produces the 70.3% line and 57.3% branch coverage figures for `SafetyProto.Shared`. The fourth and fifth commands reproduce the CLI session summaries reported in Table 2: 5/5 tasks and 750 points for PPE selection, and 9/9 tasks and 1,400 points for PPE inspection. The sixth command runs the EventBus benchmark procedure reported in Table 6; because timing results depend on the host machine, the repository also retains the raw output used for the reported values. The repository's measurement record identifies the Git ranges and counting procedures used for Tables 3 and 4. The full CLI run and `SessionIntegrationTests` load the same `default.json` used by Unity; Unity ignores its optional scripted-input field. The Unity project and a Meta Quest 3 are required to reproduce on-device interaction behaviour; no on-device session total is inferred from the CLI result.

5.6 Performance Characterisation

To complement the structural evaluation, the dispatch path of the `HarnessEventBus` — the engine-independent implementation exercised by the CLI harness — was benchmarked on .NET 10 on an AMD Ryzen 7 3800X with Windows 11. Each metric is the median of five runs of 1 M iterations, each

Table 4. Change-impact evidence for selected architectural claims.

Change	Measured impact	Architectural reading
Shared JSON data layer and authoring tool	79 source files, 2,293 insertions, 885 deletions; net +1,408 LoC	Change concentrated in the shared scenario model, its consumers, runtime actions, and authoring tool.
Meta XR SDK 85.0.0 → 201.0.0	8 files, 668 insertions, 55 deletions; 0 C# files	SDK churn was confined to packages, project settings, Android manifest, and scene wiring; `Domain` changed 0 files.
`SessionLoggerCore` addition	7 files, 264 insertions, 172 deletions	Logging was added through bus subscriptions in `Core` and `Utils`; gameplay producers were unchanged.
Scenario authoring or swap	0 C# source changes; external JSON overrides require no application rebuild	Training content changes within the existing capability catalog are data edits rather than logic changes.

Table 5. Architectural claims and supporting evidence.

Claim	Evidence
Engine independence	`Domain.Core.asmdef` sets `noEngineReferences:true`; 16 Domain files and 2,172 LoC compile in pure .NET through `SafetyProto.Shared`; `Domain.Core` has $I = 0.17$, `EventBus.Core` has $I = 0.00$, and `SafetyProto.Shared` has $I = 0.00$.
SDK isolation	The Meta XR SDK update confined its changes to configuration and scene-wiring files, with zero C# and zero `Domain` or `Networking` changes (Table 4); the 17 Meta-XR-importing files are all outside `Domain`.
Testability outside Unity	46 tests pass through `dotnet test` without the Unity Editor; 8 integration tests exercise the domain stack; coverage of the shared library (`SafetyProto.Shared`) is 70.3% line and 57.3% branch.
Zero-modification scenario authoring	Runtime scenario data is JSON; authoring or swapping a scenario is a data edit with no C# changes and no recompilation (Table 4); load-validation tests cover malformed input.
Event-driven decoupling	The prototype defines 16 typed bus channels and 13 payload types; `SessionLoggerCore` was added without modifying gameplay producers (Table 4); the module graph is a strict layered DAG with no cycles.
Correct safety and scoring semantics	Integration tests cover group-timeout terminality, PPE-missing penalty application, event ordering, dependency gating, and scenario-load validation.

preceded by a discarded 10 000-iteration warm-up; measurements use `System.Diagnostics.Stopwatch` with no external benchmarking framework. Table 6 reports the results.

Table 6. EventBus dispatch characterisation on .NET 10 (median of five 1 M-iteration runs).

Metric	Value
Dispatch latency (1 subscriber)	0.14 µs/event
Throughput	7,026,454 events/sec
Latency at 10 subscribers	0.16 µs/event (0.99×)
Latency at 100 subscribers	0.41 µs/event (2.53×)
Latency at 1 000 subscribers	2.70 µs/event (16.62×)
Memory overhead (13 payload classes, 10 handlers each)	31.3 KB

Dispatch is sub-microsecond at the prototype's typical fan-out (fewer than ten handlers per event type): a single user action that cascades into four to five events consumes well under one microsecond of bus time, three orders of magnitude below the 2 ms per-frame budget enforced by `EventBusRunner` (Section 4.10). Subscriber scaling is near-flat up to N=10 and approximately linear beyond, reaching a $16.62\times$ slowdown at N=1 000 — two orders of magnitude above the prototype's typical configuration.

Two caveats apply. First, measurements were taken on the .NET runtime through the `HarnessEventBus`, not inside the Unity Player; the in-engine `EventBus` adds a frame-budget loop, `UnityEvent` hooks, and `ScriptableObject` lifecycle overhead, whose characterisation under the Unity Profiler is deferred to follow-up work. Second, the benchmark uses a single payload type (`ActionAttemptEvent`) with empty-body handlers, isolating dispatch cost from handler workload; realistic deployments will be dominated by handler logic rather than by bus dispatch.

5.7 Architectural Comparison

Table 7 contrasts the EDA architecture with a literature-derived monolithic Unity baseline across eight maintainability properties. We base the comparison on measurements from the prototype (Table 2). Monolithic designs co-locate business logic, rendering, and SDK integration in `MonoBehaviour` components, hindering isolated testing

```
=== SafetyProto CLI Harness ===
Scenario: PPEInspection
Participant: P001
Groups: 2

  14:55:33.626  SessionStarted
--- Transcript ---
  14:55:33.643  GroupStarted       | Selecao de EPIs
  14:55:33.646  TaskStarted        | Equipar Botina de
      Seguranca
  14:55:33.866  PpeStateChanged    | Boots=WORN
  14:55:33.870  TaskCompleted      | Equipar Botina de
      Seguranca
  14:55:33.870  ScoreChanged       | Delta=100, Total
      =100
  [... 4 further PPE equip cycles, structurally
      identical ...]
  14:55:35.005  GroupCompleted     | Selecao de EPIs
  14:55:35.006  GroupStarted       | Inspecao em
      Andaime Fachadeiro
  14:55:35.006  TaskStarted        | Conectar Talabarte
       ao Ponto de Ancoragem
  14:55:35.208  ActionAttempt      | connect_harness
  14:55:35.208  TaskCompleted      | Conectar Talabarte
       ao Ponto de Ancoragem
  14:55:35.208  ScoreChanged       | Delta=200, Total
      =950
  [... 2 installation tasks ...]
  14:55:35.629  ActionAttempt      | flag_safety_net
  14:55:35.630  TaskCompleted      | Reportar
      Irregularidade na Tela Fachadeira
  14:55:35.630  ScoreChanged       | Delta=100, Total
      =1400
  14:55:35.832  SessionCompleted   | Time=2.10s, Score
      =1400, Tasks=9/9
------------------
Session summary: 9/9 tasks, score 1400, 2.10s
```

Figure 6. Excerpt from the CLI harness transcript of the nine-task scenario, executed without graphics, headset, or Unity.

```
pwsh ./scripts/reproduce-manuscript-evidence.ps1

# macOS/Linux equivalent:
./scripts/reproduce-manuscript-evidence.sh

dotnet test Tools/SafetyProto.Tests/SafetyProto.Tests.
    csproj

dotnet test Tools/SafetyProto.Tests/SafetyProto.Tests.
    csproj \
  --filter FullyQualifiedName~SessionIntegrationTests

dotnet test Tools/SafetyProto.Tests/SafetyProto.Tests.
    csproj \
  --collect:"XPlat Code Coverage" \
  --settings Tools/SafetyProto.Tests/coverlet.
      runsettings

dotnet run --project Tools/CliHarness -- \
  Tools/CliHarness/scenarios/ppe_equip.json

dotnet run --project Tools/CliHarness -- \
  Assets/_SafetyProto/Resources/Scenarios/default.json

dotnet run --project Tools/EventBusBench
```

Figure 7. Commands that reproduce the reported evidence, run from the repository root. The first two are wrapper scripts that execute the whole sequence; the remaining commands reproduce each result individually.

and making SDK migrations invasive. The monolithic column characterises the pattern predominant in the reviewed VR safety-training literature, not a replicated implementation of the same scenario; Section 6.1 discusses the limits of this comparison. The EDA architecture confines the 17 Meta XR SDK imports to engine-facing code outside `Domain`; the Domain Core (16 files, 2,172 LoC) compiles through `SafetyProto.Shared` without `UnityEngine` dependencies. This separation enabled two outcomes: (i) 46 tests execute outside Unity, and (ii) the CLI harness completed a nine-task session without Unity. Adding `SessionLoggerCore` illustrates the open/closed property: the change touched seven files in `Core` and `Utils`, with 264 insertions and 172 deletions, while gameplay producers were unchanged.

5.8 Observed Limitations

The EDA architecture introduces trade-offs:

- **Debugging complexity** — growing event counts require dedicated tracing tools (e.g., an event log inspector); the `SessionLogger` output partially addresses this. Logical defects at the event dispatch level (e.g., multiple handlers for one type) produce no errors or exceptions, surfacing as silent failures. Core/Mono separation and out-of-engine unit tests make these defects diagnosable without Unity;
- **Event taxonomy discipline** — lacking upfront event hierarchy design, proliferating fine-grained events can create implicit coupling via shared payload assumptions;
- **High-frequency event latency** — the `EventBus` is synchronous; high-frequency sensor polling (e.g., continuous hand updates) may require asynchronous dispatch or a message broker;
- **Log structural integrity requires verification** — final scores and task counts can remain correct despite log anomalies like duplicated entries or missing transitions. Analyses based on event sequences require log validation to avoid contamination. We recommend schema validation before ingesting logs;
- **Higher initial complexity** — EDA is more complex than monolithic designs for trivial systems; benefits scale with complexity.

6 Discussion

The results in Section 5 support our central thesis: an event-driven architecture is a viable foundation for immersive occupational safety training systems, addressing limitations documented in the literature.

Connection to literature gaps. Akindele et al. [Akindele *et al*., 2024] identify personalisation, adaptability, and modularity as unmet requirements in VR safety training. Our architecture addresses personalisation via an engine-neutral JSON data layer, allowing safety engineers to author scenarios without modifying C# code. It addresses adaptability through input substitution: the CLI evaluation replaced the XR input adapter with a console reader without downstream code changes, illustrating the substitution mechanism that a migration between XR devices would exercise. It addresses modularity empirically; we added `SessionLogger` without modifying gameplay modules, and 31 shared-library files compile outside Unity through `SafetyProto.Shared`. Following Obukhov et al. [Obukhov *et al*., 2022], who propose architectural decomposition to reduce coupling in virtual training, our architecture instantiates this in occupational safety using a typed event protocol and independently testable cores.

Table 7. Analytical comparison between a literature-derived monolithic Unity baseline and the proposed event-driven architecture, based on the implemented prototype.

Property	Monolithic (typical)	Proposed (EDA)
Engine independence	Logic embedded in `MonoBehaviour` lifecycle; requires Unity.	Domain Core (16 files, 2,172 LoC) has zero `UnityEngine` dependencies and compiles through `SafetyProto.Shared`.
Testability	Testing requires mocking the engine or running Play-mode tests.	46 tests run outside Unity; CLI harness exercises the nine-task scenario without a graphics engine.
SDK isolation	XR SDK calls (`OVRInput`, `Oculus.Interaction`) spread across modules.	Meta XR SDK imports appear in 17 engine-facing files; `Domain` contains zero SDK references.
Scenario authoring	Adding a scenario requires modifying C# source files and recompiling.	New scenarios authored as JSON files through the desktop tool or by hand; zero C# changes required.
Adding an evaluation module	New logic requires modifying gameplay modules to expose state.	`SessionLoggerCore` added by subscribing to `EventBus` channels; gameplay producers were unchanged.
Input modality substitution	Switching input requires changes in logic, state, and UI.	CLI harness replaces the XR adapter with a console reader; downstream modules execute unchanged.
Inter-module coupling	Modules reference each other directly; changes propagate.	Communication occurs through 16 typed `EventBus` channels; producers and consumers hold no mutual references.
CI/CD integration	Testing requires a Unity licence and a GPU build agent.	Shared library and CLI harness run on any .NET CI agent without Unity or a GPU.

Input substitutability. Confining all Meta XR SDK input calls to the Publisher and Adapter layers enables hand tracking as the primary input modality. In the tracked SDK migration, changes remained in these layers and configuration files; the Domain Core, Evaluation, and UI source files were unaffected. The CLI harness evaluation confirmed this separation, as replacing the XR input adapter required no downstream module changes.

Engine-independent validation as a transferable methodology. The CLI harness demonstrates a validation paradigm extending beyond construction safety. By exposing business logic through a typed event protocol (`IEventBus` and shared payloads), the architecture enables deterministic, engine-independent execution of the training logic stack. This provides a reproducible validation method for XR systems whose logical correctness can be evaluated independently of perceptual fidelity. Few documented XR training architectures explicitly demonstrate business-logic execution outside the immersive runtime. The same evaluation method can be tested in domains such as clinical procedure training, industrial assembly verification, or emergency response simulation.

Event fan-out as a design trade-off. The event catalog (Table 1) contains 18 logical events. A single user action, like equipping PPE, cascades into four to five events: `PpeStateChanged`, `ActionAttempt`, `TaskCompleted`, `ScoreChanged`, and optionally `SafetyViolation`. This fan-out represents a design trade-off: independent event consumption by logging, scoring, and UI subsystems eliminates inter-module coupling but increases event volume. The CLI harness completed the nine-task scripted session; its elapsed time depends primarily on the scripted actor's configured delays and is not used as a domain-processing benchmark. A distributed multi-user deployment would require batching or priority queuing.

Scenario authoring without programmer involvement. Several reviewed VR safety applications are closed implementations, requiring programmers for content expansion. The proposed data layer removes this bottleneck for scenario composition through three complementary authoring affordances. Scenario files are plain JSON documents edited through the desktop authoring tool or by hand, so adding a task or task group does not require scripting or project configuration. The authoring interface groups task identity, logic, scoring, equipment requirements, in-scenario guidance, and post-session feedback in form sections aimed at non-technical authors. Before a scenario is run, validation checks both the JSON structure and the registry of allowed interactions, reporting duplicated identifiers, naming violations, and references to undefined interactions. Together, these affordances allow safety engineers to define task sequences, specify required PPE, set scoring weights, and author feedback outside Unity, then load the resulting schema-valid file in a host without C# modification; CLI execution additionally requires scripted-input fields. Extending the interaction vocabulary itself (new action types, PPE models, or 3D scenes) remains developer and artist work in Unity, as stated in Section 4.2. This directly addresses the non-programmer authoring gap documented by Akindele *et al*. [2024], enabling domain specialists to author

and validate scenarios without programmer involvement.

Comparison with verified XR EDA frameworks. NursingXR [Obeid *et al*., 2025] modularises clinical procedures in XR using event-driven design; our work applies this principle to NR-18 and NR-35 construction safety procedures, adding a scenario exporter and an engine-independent validation layer. López et al. [López *et al*., 2021] prioritise events to preserve input responsiveness under mixed-reality real-time constraints. Our architecture addresses the complementary problem of preserving the causal and structural correctness of business events. Formally, it enforces a causal ordering constraint on core event sequences: for any task resolution cycle, the invariant `ActionAttempt` → `TaskCompleted` → `ScoreChanged` must hold. During CLI evaluation, naive synchronous dispatch logged a derived `TaskCompleted` event before its originating `ActionAttempt`, demonstrating that typed publish/subscribe alone does not guarantee correct causal ordering. Thus, our incremental contribution is twofold: we treat regulatory compliance as first-class architectural constraints, and we provide a transferable shared-library-plus-CLI method for engine-independent verification of XR business logic.

Central argument. Event-driven decoupling provides practical advantages for XR training systems requiring extensibility, engine-independent validation, and SDK isolation. The CLI harness refined this claim: rather than asserting that the exact same code executes across runtimes (which is partially untrue for physics-dependent modules like `PPEManager`), the architecture instead exposes a well-defined event protocol that any participant may implement. The CLI harness, AI observers, external script injectors, and remote evaluators all instantiate this protocol and integrate without modifying existing modules. This protocol extensibility allows the system to evolve without architectural rewrites, proving more valuable than strict source-level portability.

6.1 Threats to Validity

We acknowledge three threats to validity.

Single-prototype evaluation. Our architectural claims rely on a single prototype targeting construction safety under NR-18 and NR-35. Although the event protocol and shared-library structure are domain-agnostic, their transferability to other regulatory contexts (e.g., industrial hygiene, electrical safety) lacks empirical demonstration. Furthermore, dispatch characterisation is reported on the .NET runtime (Section 5.6, Table 6), not inside the Unity Player. In-engine measurement of the same path — including frame-budget interaction, `UnityEvent` hooks, and behaviour under multi-user or high-frequency loads — is deferred to follow-up work.

Inferred baseline comparison. The monolithic baseline in Table 7 represents the predominant architecture in the VR training literature, not a replicated monolithic implementation of the same scenario. A controlled comparison measuring maintenance effort, defect rates, and modification cost across identical scenarios in both architectures would provide stronger evidence, but fell outside our scope.

No deployment-scale validation. The development team and automated test harnesses exercised the prototype, but it has not been deployed in an active construction worker training programme. Consequently, non-programmer scenario authoring and long-term maintainability remain architectural affordances rather than empirically confirmed outcomes.

7 Conclusion

This case study demonstrates the feasibility of decoupling XR business logic from a game engine and validating it deterministically in standard CI environments, using an event-driven modular architecture instantiated in a VR safety training prototype for Brazilian regulatory norms NR-18 and NR-35.

The central architectural contribution is a typed `EventBus` that decouples gameplay, evaluation, and UI layers, combined with an engine-neutral JSON data layer enabling scenario authoring without code changes. We evaluated the architecture using an engine-independent unit test suite and CLI harness, providing empirical evidence of the business logic layer's engine-independence.

The evaluation supports four properties in the prototype: (i) new scenarios were authored without logic module changes; (ii) XR SDK updates remained confined to the Publisher and Adapter layers, leaving the Domain Core, Evaluation, and UI layers unaffected; (iii) the business logic stack executed outside the graphics engine, enabling CI automated testing; and (iv) the `SessionLogger` was integrated without modifying existing modules, consistent with the open/closed property of the event-driven design.

These findings address a literature gap by proposing a replicable, extensible software architecture for VR safety training potentially applicable to other regulatory contexts.

Future Work

Future directions include:

- **Multi-user support** via a distributed implementation of the existing `IEventBus` port (`Subscribe`, `Unsubscribe`, and `Publish<T>`). The current code already has three interchangeable implementations of this port: the Unity `EventBus` ScriptableObject with queued frame-flushed delivery, the synchronous CLI `HarnessEventBus`, and `FakeEventBus` for tests. A server-backed bus for multiplayer or external-system integration would attach at this seam;
- **Migration to Unity DOTS / ECS** — adapting the dispatch model for high-frequency streams (continuous gesture recognition, multi-sensor input fusion). The engine-independent Domain Core allows replacing the `EventBus` without logic layer modifications;
- **BIM and Digital Twin integration** — connecting real-time IoT sensor feeds to enable training grounded in live site data;
- **Automated scenario authoring via MCP and LLM agents** — using a Model Context Protocol (MCP)[1] server to generate JSON scenario definitions without programmer involvement;

[1] Model Context Protocol: `https://modelcontextprotocol.io`

- **AI-driven virtual instructor** — subscribing a language model to the `EventBus` for natural language adaptive guidance;
- **Authoring tooling** — extending the desktop editor beyond task-scenario composition, so that registering new 3D scenes, PPE models, and interaction capabilities no longer requires developer or artist intervention in Unity;
- **Deployment and authoring studies** — deploying the prototype in an active worker-training programme and observing safety engineers authoring scenarios with the desktop tool, so that non-programmer authoring and long-term maintainability become measured outcomes rather than architectural affordances;
- **Controlled comparative evaluation** — implementing the same scenario in a monolithic baseline and in a second training domain to measure maintenance effort, defect rates, and generality beyond this case study.

The core design principle — verifying business logic independently of its rendering engine — suggests a transferable model for immersive training systems and other XR systems engineering contexts in which what must be certified is the rules a session enforces, not the fidelity with which they are rendered.

Declarations

Authors' Contributions

Following the CRediT taxonomy [NISO, 2024]: **Abner Augusto R. M. A. de Souza:** Conceptualization, Methodology, Software, Validation, Investigation, Data Curation, Writing – Original Draft, Visualization. **José de Paula Barros Neto:** Supervision. **Creto Augusto Vidal:** Conceptualization, Supervision, Writing – Review & Editing. **Joaquim Bento Cavalcante Neto:** Conceptualization, Methodology, Supervision, Writing – Review & Editing, Project Administration.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

The authors acknowledge the Universidade Federal do Ceará (UFC) for providing the institutional infrastructure used in this research, and the Computer Graphics, Virtual Reality and Animations research group (CRAb/UFC) for the resources and support provided during prototype development.

Funding

This research received no external funding.

Availability of data and materials

The source code of the prototype, including the shared library, CLI harness, unit test suite, and all JSON scenario definitions, is publicly available at `https://github.com/abner-augusto/safety-training-proto`. The immutable snapshot used for the reported evaluation is archived as repository release `v1.0.0` (commit `d8d16d2`).

Use of Generative AI Tools

In accordance with CNPq Ordinance No. 2,664/2026, the authors disclose the use of generative AI tools during the preparation of this manuscript. Claude (Anthropic) was used to assist with bibliography verification and LaTeX formatting. All generated content was critically reviewed, verified, and edited by the authors, who assume full responsibility for the accuracy and originality of the final text. No AI tool was used to generate research results, architectural design decisions, or source code of the prototype.

References

Abotaleb, I., Hosny, O., Nassar, M., Bader, S., Elrifaee, A., Ibrahim, M., El Hakim, M., and Sherif, M. (2022). An interactive virtual reality model for enhancing safety training in construction education. *Computer Applications in Engineering Education*, 31(2):324–345. DOI: 10.1002/cae.22585.

Ackermann, P. (2023). AR patterns: Event-driven design patterns in creating augmented reality experiences. In *Virtual Reality and Mixed Reality: 20th EuroXR International Conference (EuroXR 2023)*, volume 14410 of *Lecture Notes in Computer Science*, pages 93–114, Cham. Springer. DOI: 10.1007/978-3-031-48495-7_6.

Akindele, N., Taiwo, R., Sarvari, H., Oluleye, B. I., Awodele, I. A., and Olaniran, T. O. (2024). A state-of-the-art analysis of virtual reality applications in construction health and safety. *Results in Engineering*, 23:102382. DOI: 10.1016/j.rineng.2024.102382.

Alzarrad, A. A., Miller, M., Durham, L., and Chowdhury, S. (2024). Revolutionizing construction safety: introducing a virtual reality interactive system for training US construction workers to mitigate fall hazards. *Frontiers in Built Environment*, 3:1320175. DOI: 10.3389/fbuil.2024.1320175.

Associação Brasileira de Normas Técnicas (2024). ABNT NBR 16325-1: Proteção contra quedas de altura — dispositivos de ancoragem — parte 1: Requisitos e métodos de ensaio [protection against falls from heights — anchorage devices — part 1: Requirements and test methods]. Technical report, ABNT. Available at: `https://abnt.org.br/`.

Brooke, J. (1996). SUS: A 'quick and dirty' usability scale. In Jordan, P. W., Thomas, B., Weerdmeester, B. A., and McClelland, A. L., editors, *Usability Evaluation in Industry*, pages 189–194. Taylor and Francis, London. DOI: 10.1201/9781498710411-35.

Castet, E., Termoz-Masson, J., Vizcay, S., Delachambre, J., Myrodia, V., Aguilar, C., Matonti, F., and Kornprobst, P. (2024). PTVR – a software in Python to make virtual reality experiments easier to build and more reproducible. *Journal of Vision*, 24(4):19. DOI: 10.1167/jov.24.4.19.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131. DOI: 10.1145/857076.857078.

Gazis, A. and Katsiri, E. (2024). E-polis: An innovative and fun way to gamify sociological research with an educational serious game – game development middleware approach. *International Journal of Education and Information Technologies*, 18:20–32. DOI: 10.46300/9109.2024.18.3.

Guo, X., Liu, Y., Tan, Y., Xia, Z., and Fu, H. (2024). Hazard identification performance comparison between virtual reality and traditional construction safety training modes for different learning style individuals. *Safety Science*, 180:106644. DOI: 10.1016/j.ssci.2024.106644.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1):75–105. DOI: 10.2307/25148625.

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, Deploying Messaging Solutions*. Addison-Wesley Professional. Book.

Isingizwe, F., Eiris, M., and Jalil Al-Bayati, A. (2024). Enhancing safety training engagement through immersive storytelling: A case study of construction safety in virtual reality. *Safety Science*, 179:106631. DOI: 10.1016/j.ssci.2024.106631.

Joshi, S., Hamilton, M., Warren, R., Faucett, D., Tian, W., Wang, Y., and Ma, J. (2021). Implementing virtual reality technology for safety training in the precast/prestressed concrete industry. *Applied Ergonomics*, 90:103286. DOI: 10.1016/j.apergo.2020.103286.

Kennedy, R. S., Lane, N. E., Berbaum, K. S., and Lilienthal, M. G. (1993). Simulator sickness questionnaire: An enhanced method for quantifying simulator sickness. *The International Journal of Aviation Psychology*, 3(3):203–220. DOI: 10.1207/s15327108ijap0303_3.

López, M., Terrón-Ibáñez, S., Lombardo, J., and Gonzalez Crespo, R. (2021). Towards a solution to create, test and publish mixed reality experiences for occupational safety and health learning: Training-MR. *International Journal of Interactive Multimedia and Artificial Intelligence*, 7:212–223. DOI: 10.9781/ijimai.2021.07.003.

Man, S. S., Wen, H., and So, B. C. L. (2024). Are virtual reality applications effective for construction safety training and education? A systematic review and meta-analysis. *Journal of Safety Research*, 88:230–243. DOI: 10.1016/j.jsr.2023.11.011.

Martusciello, F., Muccini, H., and Forgione, A. (2025). Gamified AR and supervised AI for cultural heritage: The amiternum site experience. In *Proceedings of the TC26 MetroArcheo Conference 2025 on Metrology for Cultural Heritage*, pages 521–526, Bergamo, Italy. IMEKO. DOI: 10.21014/tc26-2025.097.

Ministério do Trabalho e Emprego (2021). Norma regulamentadora 35 — trabalho em altura, portaria mte 735, de 6 de outubro de 2021 [regulatory standard no. 35 — work at heights]. Technical report, Ministério do Trabalho e Emprego, Governo Federal do Brasil. Available at: `https://www.gov.br/trabalho-e-emprego/pt-br/acesso-a-informacao/participacao-social/conselhos-e-orgaos-colegiados/comissao-tripartite-partitaria-permanente/normas-regulamentadora/normas-regulamentadoras-vigentes/norma-regulamentadora-no-35-nr-35`.

Ministério do Trabalho e Emprego (2022). Norma regulamentador 18 — segurança e saúde no trabalho na indústria da construção portaria sit 163, de 27 de junho de 2022 [regulatory standard no. 18 — occupational safety and health in the construction industry]. Technical report, Ministério do Trabalho e Emprego, Governo Federal do Brasil. Available at: `https://www.gov.br/trabalho-e-emprego/pt-br/acesso-a-informacao/participacao-social/conselhos-e-orgaos-colegiados/comissao-tripartite-partitaria-permanente/normas-regulamentadora/normas-regulamentadoras-vigentes/norma-regulamentadora-no-18-nr-18`.

Ministério Público do Trabalho and Organização Internacional do Trabalho (2024). Observatório de segurança e saúde no trabalho – SmartLab [observatory of occupational safety and health — smartlab]. `https://smartlabbr.org/sst`. Accessed: 2026-04-16.

Moreira, F. G. P., de Oliveira, C. P., and Farias, C. A. (2024). Workplace accidents and the probabilities of injuries occurring in the civil construction industry in Brazilian Amazon: A descriptive and inferential analysis. *Safety Science*, 174:106478. DOI: 10.1016/j.ssci.2024.106478.

NISO (2024). CRediT – contributor roles taxonomy. `https://credit.niso.org`. ANSI/NISO Z39.104-2022. Accessed: 2026-05-02.

Nykänen, M., Teperi, A.-M., *et al*. (2020). Implementing and evaluating novel safety training methods for construction sector workers: Results of a randomized controlled trial. *Journal of Safety Research*, 75:205–221. DOI: 10.1016/j.jsr.2020.09.015.

Obeid, M. F., Ewais, A., and Asia, M. R. (2025). NursingXR: Advancing nursing education through virtual reality-based training. *Applied Sciences*, 15(6):2949. DOI: 10.3390/app15062949.

Obukhov, A. D., Volkov, A. A., and Nazarova, A. O. (2022). Microservice architecture of virtual training complexes. *Informatics and Automation*, 21(6):1265–1289. DOI: 10.15622/ia.21.6.7.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77. DOI: 10.2753/MIS0742-1222240302.

Quamara, M., Capra, L., Villani, V., Carlevaro, C., Celiktutan, O., Peretti, A., Piazzola, M., Ruo, A., Sabattini, L., Schmuck, V., and Viganò, L. (2024). Towards a modular architecture for eXtended reality systems. In *8th International Conference on Artificial Intelligence and Virtual Reality*. Springer. DOI: 10.1007/978-981-96-1154-6_17.

Scorgie, D., Feng, Z., Paes, D., Parisi, F., Yiu, T. W., and Lovreglio, R. (2024). Virtual reality for safety training: A systematic literature review and meta-analysis. *Safety Science*, 171:106372. DOI: 10.1016/j.ssci.2023.106372.

Simões, B., De Amicis, R., Barandiaran, I., and Posada, J. (2018). X-Reality system architecture for industry 4.0 processes. *Multimodal Technologies and Interaction*, 2(4):72.

DOI: 10.3390/mti2040072.

Witmer, B. G. and Singer, M. J. (1998). Measuring presence in virtual environments: A presence questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240. DOI: 10.1162/105474698565686.

Xu, Z. and Zheng, N. (2021). Incorporating virtual reality technology in safety training solution for construction site of urban cities. *Sustainability*, 13(1):243. DOI: 10.3390/su13010243.