UNIVERSITY OF CALGARY

Evaluating LLM-Generated Test Scripts, Manual Test Development, and Capture-and-Replay Approaches for Testing XR Applications

by

Seyed Nami Modarressi

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA
AUGUST, 2025

Abstract

The rapid advancement of Extended Reality (XR) technologies, driven by substantial improvements in hardware and software, has accelerated the development and deployment of XR applications. A critical yet challenging stage in software development is testing, which is often labor-intensive and time-consuming. As a result, testing may be partially or entirely overlooked, leading to potential issues.

To address these challenges, various software testing methodologies and tools have been developed to streamline the process. This research introduces a generative AI-based testing tool, TestGPT, specifically designed for Unity 3D applications. The tool automatically generates test scripts based on its analysis of Unity 3D scenes provided by testers and test scenarios written in natural language.

To evaluate our proposed approach, we conducted an empirical study with 15 participants who had experience in Unity development, comparing three testing methods: Manual Test Development, Capture-and-Replay, and our generative AI approach. All participants applied each method to a sample project across multiple testing scenarios to assess their strengths and limitations. Quantitative performance metrics and qualitative user feedback were systematically analyzed.

The results show that Manual Test Development offers high control and precision but lacks scalability for large projects; TestGPT is efficient for simple scenarios but less effective in complex contexts, especially without retaining the memory of previously generated test scripts; and Capture-and-Replay is intuitive and well-suited for regression testing but sensitive to UI changes.

These findings highlight the trade-offs between flexibility, efficiency, and adaptability, providing practical insights into the suitability of each method for XR application testing.

Acknowledgements

I would like to express my profound gratitude to my supervisor, Dr. Frank Maurer, for his invaluable guidance, unwavering support and encouragement throughout my research. His expertise and insights were instrumental in the success of this thesis.

I also want to extend my heartfelt gratitude to my father, mother, and brother for their love, unwavering encouragement, unyielding support, patience, and the many sacrifices they have made throughout my life. They have always inspired me to push boundaries and strive for excellence in all my endeavors. I also want to remember my grandmother, who is no longer with us to witness my graduation.

I am grateful to all the important people in my life, all my friends, and labmates. Their constant presence and unwavering support have been invaluable to me. Your contributions have enriched my experience and have made a substantial impact on my personal and professional growth.

At the end of the day, the person who changed my career and brought me to this field is the one who truly changed my life. The patient and caring nature of Dr. Jahangir made all the difference. Thank you, Dr. Jahangir.

To my parents, who despite the huge time difference, have always been a constant source of support and inspiration. I dedicate this thesis to them, in recognition of their unwavering love and encouragement throughout my academic journey. Their belief in me and constant support have been a driving force behind my success. This thesis is a testament to their sacrifices and dedication to my education despite the distance. Thank you for everything.

Table of Contents

List of Figures

List of Tables

Abbreviations

Symbol	Definition
XR	Extended Reality
HMD	Head-Mounted Display
AI	Artificial Intelligence
LLM	Large Language Model
QA	Quality Assurance
UWP	Universal Windows Platform

Chapter 1

Introduction

The global market for XR has experienced growth in recent years, with continued evolution driven by substantial investments from major technology companies. In 2023, Apple introduced Vision Pro, powered by Vision OS, a dedicated XR operating system designed specifically for its HMD headset [1]. Similarly, in late 2024, Google announced Android XR, an XR operating system designed for next-generation computing experiences [2]. These developments, along with the progress of other companies like Meta in releasing powerful yet reasonably priced headsets such as the Meta Quest 3 [3] or Meta Quest 3s [4], and in developing APIs and tools for creating more immersive experiences [5], may represent a pivotal shift in the XR landscape, potentially accelerating mainstream adoption. As XR technologies continue to expand, the development of complex XR applications is increasing, further facilitated by advancements in HMDs.

With XR applications becoming increasingly prevalent across various critical domains [8] and numerous platforms and devices, their development and testing processes have consequently become more complex [6, 8]. Software testing, a crucial phase in software development, can be time-consuming, expensive, and resource-intensive [68, 69]. Often, due to these challenges, testing phases are overlooked, potentially leading to serious issues including user safety risks and significant financial losses for companies [70]. To address these challenges, software testing methodologies have been developed to streamline and optimize the testing process.

XR applications inherently possess unique characteristics that set them apart from traditional applications, such as standard mobile 2D apps. While traditional applications typically rely on flat user interfaces, simple touch or mouse interactions, and predictable screen-based workflows, XR applications operate in immersive 3D environments and often require spatial awareness, real-time sensor input, and natural user interactions. These may include selecting and manipulating virtual objects, navigating virtual or mixed envi-

ronments, and performing coordinated tasks within a dynamic spatial context [7, 8]. Such features introduce challenges not typically present in conventional apps, making the testing process for XR applications more complex and demanding [8].

Traditional software testing approaches include manual test development, in which testers explicitly write and code detailed test scripts for systematic verification; manual testing, where testers conceptually design and directly execute test cases without coding or automation; and capture-and-replay, which records and replays user actions to validate system behavior. Additional methods, such as exploratory testing, model-based testing, and behavior-driven development, offer alternative strategies for identifying defects and improving coverage. Manual methods allow for flexibility, while automated and model-based techniques enhance scalability, repeatability, and efficiency [12, 13].

Meanwhile, advancements in generative AI, particularly its enhanced capability in code generation, have created new opportunities for software testing by enabling automated generation of test scripts. Leveraging this potential, this thesis introduces a tool called *TestGPT*, designed to create test scripts based on Unity scene information and natural-language test scenarios.

This study aims to evaluate the performance and usability of the proposed tool and compare it with two testing methods: capture-and-replay and manual test development. The objective of this research is to identify effective and practical testing methodologies for XR applications.

1.1 Motivation

With the rapid growth of XR technologies and their expanding integration into various industries, including healthcare, education, entertainment, and industrial training, the demand for high-quality, stable, and responsive XR applications has increased [8]. Ensuring the reliability and performance of these applications is critical, as even minor bugs or interaction failures can degrade user experience and reduce the overall effectiveness of the application. Consequently, software testing has become a crucial component of the XR application development lifecycle. However, testing XR applications presents unique challenges due to their immersive nature, dynamic environments, and reliance on sensor-driven interactions, which refer to user interactions in an application or system that rely on real-time data from hardware sensors to detect the user's position, movement, gestures, or environment and respond accordingly [6, 7].

Generally, in the domain of software testing, various methodologies have been developed to address these challenges, each with distinct benefits and limitations. manual test development, while flexible and intuitive, is often time-consuming and difficult to scale for complex scenarios [15, 26]. Capture-and-replay methodologies offer reproducibility but can encounter difficulties when the user interface of the application

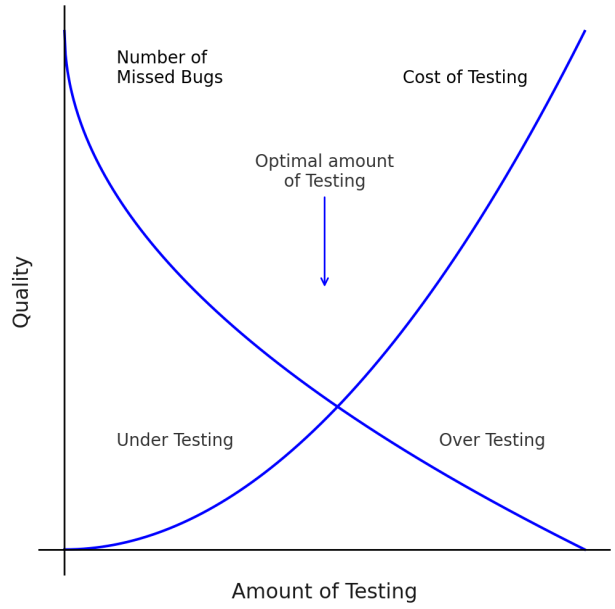

Figure 1.1: Every Software Project has optimal test effort [40].

changes [71]. As XR applications continue to grow in complexity and sophistication, the demand for more efficient and adaptive testing methodologies becomes increasingly critical [8].

To address the limitations of existing methodologies, this research explores the potential of generative AI as a way to enhance XR application testing by reducing the time required for testing while maintaining reproducibility and usability. Recent advancements in generative AI, particularly in automated code generation, suggest a promising future for the integration of such technologies into software testing. This thesis presents a tool that leverages generative AI to automate test case creation and systematically compares its effectiveness with established testing methodologies. By evaluating the strengths and weaknesses of each approach, this research seeks to determine whether generative AI can deliver benefits to software testing.

1.2 Research Goal

The primary objective of this research is to compare different software testing methodologies for XR applications developed in Unity, including capture-and-replay, manual test development, and the use of generative AI for test generation. A secondary objective is to evaluate the performance of a custom-built generative AI–based tool that utilizes scene information and test scenarios to generate automated test scripts for XR applications.

1.3 Research Question

In alignment with the objective of comparing testing methods for Unity 3D applications and analyzing the performance of using generative AI for test generation, we pose two important research questions:

How do different testing methods perform when applied to for XR applications developed in Unity applications?

Additionally, we seek to understand: *Can generative AI be effective in generating tests based on scene information and textual test scenarios provided by the tester?*

1.4 Thesis Contributions

The main contributions of this thesis are as follows:

- A systematic comparison of three testing methodologies—manual test development, capture-and-replay, and AI-assisted test creation—to identify their relative strengths, weaknesses, and suitability for different testing contexts.
- Discussion of the potential benefits and limitations of applying generative AI for XR test generation, particularly in terms of usability, efficiency, and scalability compared to traditional methods.
- Introduction and evaluation of *TestGPT*, a generative AI–based tool that automatically generates Unity test scripts from scene information and natural-language test scenarios.

1.5 Thesis Structure

Chapter One introduces the overall objectives and key concepts of the research. It outlines the motivation driving the study, defines the research goals, poses the primary research questions, and summarizes the study's contributions.

Chapter Two provides essential background information for this thesis. This includes a detailed exploration of software testing methodologies such as manual test development and capture-and-replay methods, test automation, the structure of Unity 3D applications, and generative AI and its application in code generation.

Chapter Three provides an overview of relevant prior research. It begins with a review of studies investigating the application of AI in software testing, followed by an examination of research on software testing methodologies, with particular emphasis on the capture-and-replay technique. Finally, the chapter explores

research on code generation using LLMs, assessing their effectiveness and examining related developments in the field.

Chapter Four delves into the study's discussions and overall design. It also describes the device used in the study, the results from the pilot study, participant demographic analysis, and the specific tasks and visualizations presented to participants.

Chapter Five documents the structure of Unity scene files and describes the tool developed for generating generative AI–based tests. It also covers the test automation framework and the capture-and-replay tool used in this research.

Chapter Six describes the methods used to analyze both qualitative and quantitative data and presents the results of this analysis, including numerical findings. The chapter further discusses these findings, incorporates insights from participant interviews, and presents conclusions drawn from the collected data. It also identifies the limitations encountered during the study.

Chapter Seven summarizes the research, answers the original research questions, and highlights the key findings. It also outlines possible directions for future work building on this study. The broader relevance of this research lies in advancing reliable XR testing practices applicable across diverse domains such as gaming, training, and simulation, where performance and user safety are critical.

Chapter 2

Background

2.1 Software Testing

Software testing is a critical phase in the software development lifecycle, ensuring that a system meets its specified requirements and behaves as expected [12, 13]. It not only detects bugs but also safeguards software quality, performance, and user satisfaction. Testing approaches generally include black-box testing, which validates outputs against inputs without inspecting the internal code; white-box testing, which verifies internal logic and structures; and grey-box testing, which combines both [9, 10, 11]. Tests may also be classified as functional, verifying what the system does, or non-functional, assessing how the system performs under conditions related to performance, scalability, security, and usability [72, 73]. Finally, testing can be conducted either manually or through automation, depending on the system's complexity and the goals of the testing phase.

Functional testing focuses on verifying that the software's features and capabilities operate according to the specified requirements. It evaluates what the system does, ensuring that inputs produce the expected outputs and that each function works as intended. Examples include unit testing, integration testing, system testing, and acceptance testing [72]. In contrast, non-functional testing assesses aspects of the system that relate to how it performs rather than what it does. This includes attributes such as performance, scalability, security, usability, and reliability. Non-functional testing ensures that the software not only functions correctly but also meets quality benchmarks under various conditions [73].

Manual testing involves human testers executing test cases without the aid of automation tools. Testers interact directly with the application, observe outcomes, and compare them with expected results. While manual testing can be more flexible and adaptive—especially for exploratory or usability testing—it is

often time-consuming and less efficient for repetitive tasks [12, 13]. Automated testing, on the other hand, uses scripts and specialized tools to execute tests automatically, compare actual and expected results, and generate reports. Automation is well-suited for regression testing, large-scale test coverage, and scenarios where speed, consistency, and repeatability are critical, although it requires an upfront investment in tools and script development [74, 75].

Core activities in software testing typically include test planning, test case design, execution, validation, and defect reporting. As software systems grow in scale and complexity especially in domains like web applications, mobile platforms, and interactive 3D systems software testing becomes increasingly important to ensure reliability, maintainability, and scalability of codebases [6, 8].

2.1.1 Process of Software Testing

The software testing process is a systematic set of activities aimed at ensuring that software meets its functional and non-functional requirements. While traditionally described in a linear, sequential manner—such as in the requirement analysis, test planning, test case design, execution, and closure phases—modern Agile and iterative development processes integrate these activities continuously throughout the software lifecycle [15, 16, 76].

In an Agile context, testing begins as soon as user stories and acceptance criteria are defined during sprint planning, serving a similar role to the requirement analysis phase. Test planning and test case design occur iteratively within each sprint, with testers collaborating closely with developers and product owners to refine test scenarios [76, 77]. These tests, which may include both manual and automated scripts, are maintained in a living repository that evolves alongside the product [12, 14].

Test execution is performed continuously as features are developed, often using automated pipelines that run regression and integration tests on each code commit. Manual exploratory testing is also conducted within the sprint to identify usability or edge-case issues not covered by automation. Defects are logged and resolved quickly, with re-testing and regression testing happening in the same iteration rather than in a separate phase [76, 77].

At the end of each sprint, a form of test closure occurs in the sprint review and retrospective, where the team evaluates test coverage, defect trends, and the overall quality of the increment delivered. This iterative approach ensures that testing is not a final step before release but an ongoing activity that adapts to changes in requirements, promotes rapid feedback, and supports the delivery of high-quality software in shorter cycles [13].

2.1.2 Test Automation

Test automation refers to the practice of using software tools and scripts to perform test cases automatically, reducing the need for repetitive manual test development and enabling faster, more reliable validation across software iterations [18]. It plays a vital role in modern software development, particularly in continuous integration and deployment (CI/CD) pipelines, where rapid feedback and regression checks are essential.

Automated testing enhances efficiency, repeatability, and accuracy, making it particularly useful for large, complex, or frequently updated systems. Tools like Selenium, which implement the capture-and-replay concept, are widely used to automate web application testing [19, 20]. Capture-and-replay is a testing approach in which user interactions with an application are recorded and later replayed to reproduce the same actions automatically. This technique helps reduce the need for manually re-executing repetitive test cases, making it especially useful for regression testing [8].

In other domains, such as XR and mobile environments, frameworks like iv4XR [21] and tools like TESTAR [22] automate interactions with immersive 3D scenes by simulating user behavior and validating system responses without human intervention. Similarly, Arium [17] is a framework for Unity applications that streamlines automated testing by simplifying test creation and enabling all tests to be executed automatically, without the need to run them manually one by one [24, 63].

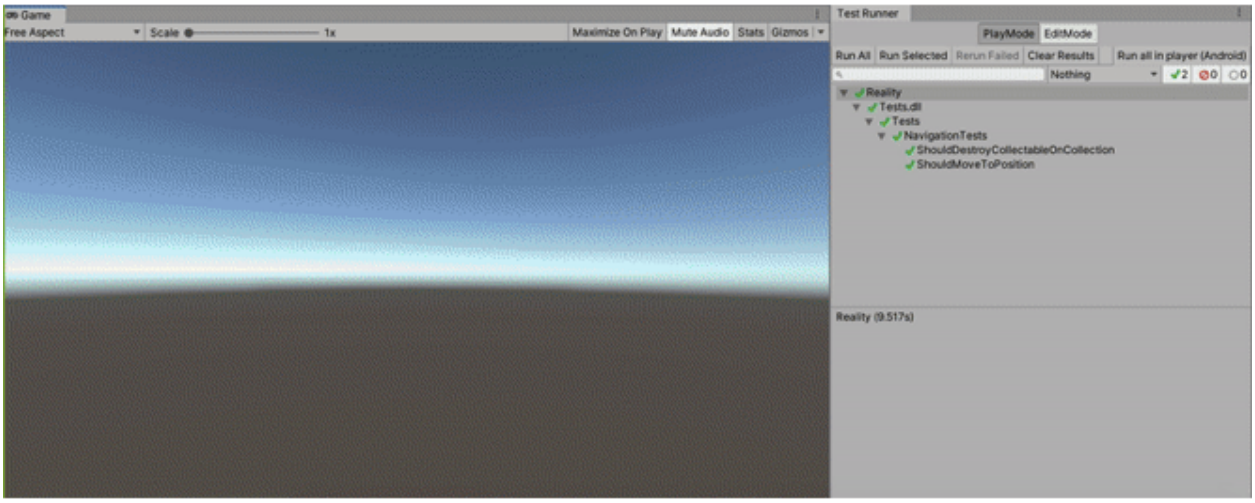

Figure 2.1: Arium is a Test Automation tool for Unity that automatically runs all tests, allowing testers to view the final results [17].

2.1.3 Software Testing Methodology

Software testing methodologies refer to structured approaches or strategies employed to plan, design, execute, and evaluate tests efficiently. These methodologies help ensure that testing is systematic, repeatable, and

aligned with project goals [78, 79].

2.1.4 Manual Test Development

Manual testing is the process of executing test cases without the use of automation tools. Testers manually interact with the software to identify bugs, verify functionality, and ensure that the application behaves as expected under various scenarios [8].

Manual test development, on the other hand, refers to writing code by hand without relying on code generators or visual tools. It allows developers full control over the logic, structure, and optimization of their programs [23].

```
[UnityTest, Order(3)]
public IEnumerator OnClickingChangeColorButtonCubeColorShouldChange()
{
    yield return new WaitForSeconds(2);
    GameObject cube = _arium.FindGameObject("red_cube");
    string changeColorButton = "ChangeColor";
    // Validate default color is red
    Assert.AreEqual(Color.red, cube.GetComponent<MeshRenderer>().material.color);
    // Click → Green
    _arium.PerformAction(new UnityPointerClick(), changeColorButton);
    yield return new WaitForSeconds(1);
    Assert.AreEqual(Color.green, cube.GetComponent<MeshRenderer>().material.color);
    // Click → Blue
    _arium.PerformAction(new UnityPointerClick(), changeColorButton);
    yield return new WaitForSeconds(1);
    Assert.AreEqual(Color.blue, cube.GetComponent<MeshRenderer>().material.color);
    yield return null;
}
```

Figure 2.2: A sample test function implemented with the Arium Testing Framework [24].

2.1.5 Capture and Replay

Capture and replay is a software testing methodology that records user interactions with an application and replays them to test the system's behavior. This approach simplifies the process of test case creation by capturing inputs, clicks, navigation paths, and expected outputs during a live session. Once recorded, these interactions can be replayed automatically during testing or after code changes, ensuring consistent behavior without needing manual repetition [25, 26, 27, 28].

One of the tools that utilizes capture and replay is Selenium IDE [30, 31]. Selenium IDE allows testers to record browser-based interactions like filling out forms, clicking buttons, and navigating through web pages. These interactions are stored as test scripts, which can then be replayed to verify test conditions.

This reduces the time and effort required for repetitive testing tasks, especially in agile environments where frequent updates are common [29].

In Selenium IDE, test conditions, often referred to as assertions or verifications, are defined to check whether the application's state matches the expected outcome at specific points in the test. For example, testers can assert that a particular text is present on the page, an element is visible, a field contains the correct value, or a button is enabled. These conditions are typically added during or after the recording process by selecting the target element and specifying the validation criteria. Assertions cause the test to fail if the expected condition is not met, while verifications log the issue but allow the test to continue, making it possible to gather more results in a single run [30].

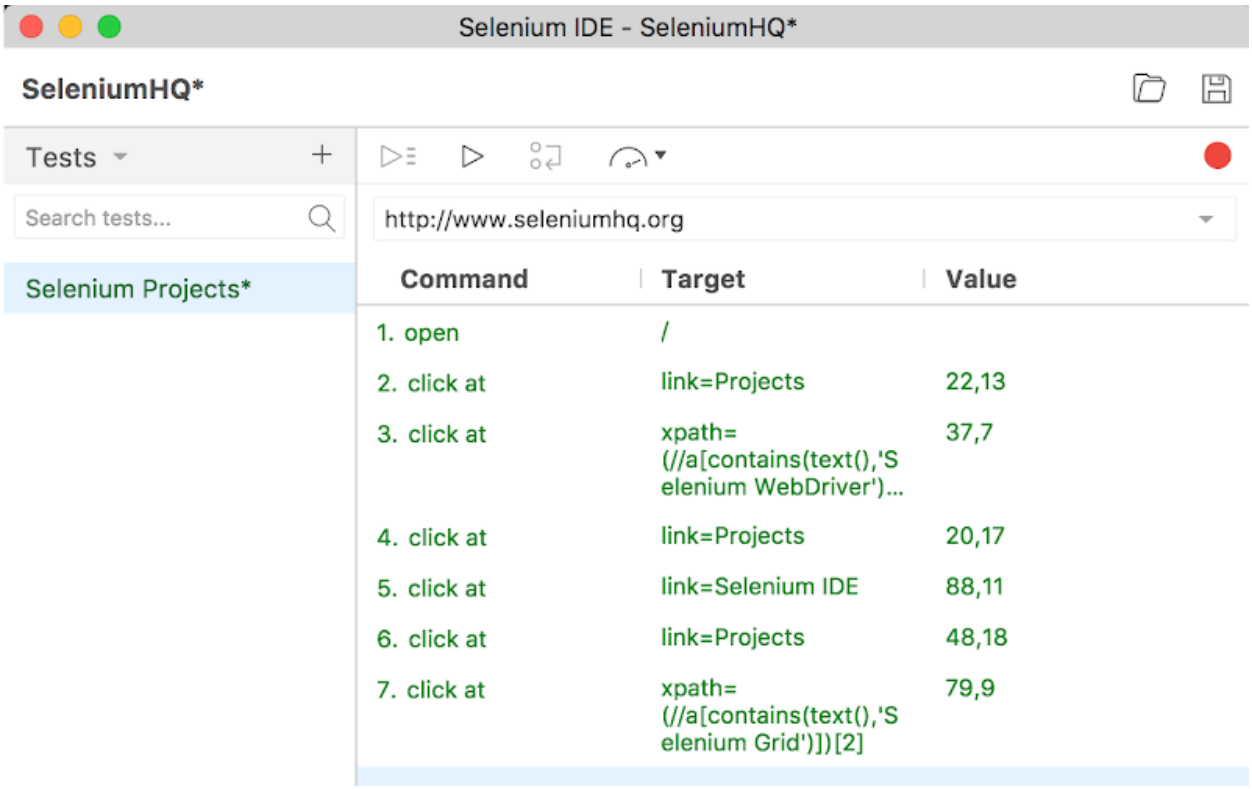

Figure 2.3: Selenium, a capture and replay testing tool for web applications [30].

2.2 XR Application Development

The rapid progress in HMD technology and the increasing availability of XR devices have significantly influenced how XR applications are developed and tested. From early VR systems that required powerful PCs and external tracking sensors to today's standalone, high-resolution, and spatially aware headsets such as the Meta Quest 3 [32] and Apple Vision Pro [1], the hardware landscape has evolved dramatically.

Modern devices offer improved ergonomics, wider fields of view, higher refresh rates, and advanced sensing capabilities such as eye, hand, and gesture tracking, enabling more natural and immersive user experiences [6, 7, 8]. These advances have also introduced new technical and testing complexities that extend beyond those encountered in traditional 2D systems.

Figure 2.4: Meta Quest 3, an example of a modern XR device [32].

XR applications are typically developed using game engines that support real-time rendering, physics simulation, and interaction design like the Unity, which provides cross-platform support and XR integration. The development process involves importing 3D models and assets, designing spatial environments, defining lighting and physics properties, and implementing user interactions through C# scripts. Developers must also consider device-specific characteristics such as tracking precision, boundary systems, and gesture or controller input [6, 7].

Unity is a powerful and versatile platform for XR development. It offers built-in support for major XR platforms through packages such as OpenXR, AR Foundation, and platform-specific SDKs, which facilitate deployment across multiple devices [33]. Unity further provides APIs for spatial anchoring, scene understanding, and input abstraction layers that unify different controller types. This allows developers to write generalized interaction logic that works across diverse XR ecosystems while leveraging the Unity Asset Store's extensive library of prefabricated assets, shaders, and plugins.

2.2.1 Structure of a Unity 3D Application

A Unity 3D scene is organized as a hierarchical collection of *GameObjects*, each representing an entity in the virtual environment. At the top level, the *Scene* contains one or more root objects, which can have nested children, forming a tree-like hierarchy. Every GameObject can hold one or more *Components* that define its behavior and visual representation—for example, a `Transform` for spatial position and orientation, a `MeshRenderer` for displaying 3D geometry, a `Collider` for physics interactions, and user-defined scripts that implement logic and interactivity. Scenes often include fundamental elements such as lighting systems, main and secondary cameras, spatial audio sources, user interface canvases, and interactive 3D objects. This modular, component-based architecture enables flexible reuse and hierarchical manipulation of objects, which is particularly advantageous for prototyping and iterative XR design.

The object hierarchy is critical in testing because it defines how objects are logically grouped, parented, and transformed relative to one another. Automated testing frameworks such as *Arium* [24, 63] rely on this hierarchy to locate GameObjects by path, tag, or component composition. This allows test scripts to verify object states, simulate user actions, and reproduce complex interaction sequences within dynamic 3D environments. However, even small structural changes—such as renaming, re-parenting, or altering component configurations—can cause test failures or misidentification of targets, emphasizing the need for consistent scene management and automated hierarchy validation.

2.2.2 Controller Input Models and Interaction Systems

Interaction in XR relies on diverse input models that translate user intent into system actions. Unity's input frameworks such as the XR Interaction Toolkit and Input System provide abstractions for various device inputs, including handheld controllers, hand tracking, eye gaze, and gesture recognition. Controllers typically offer six degrees of freedom (6DoF) tracking and multiple input modalities such as trigger, grip, and thumbstick interactions. Hand-tracking systems, by contrast, require fine-grained spatial mapping and gesture classification, enabling direct manipulation of virtual objects without physical controllers. Eye-tracking introduces another interaction layer, allowing gaze-based selection or context-aware user interfaces. Managing these diverse inputs consistently across devices is a challenge in development and testing, as each device implements input sensing and event handling differently.

2.2.3 Challenges in Testing Immersive Interactions

Testing XR applications introduces unique challenges that go beyond conventional software verification. Because interactions depend on spatial awareness, environmental context, and continuous sensor feedback,

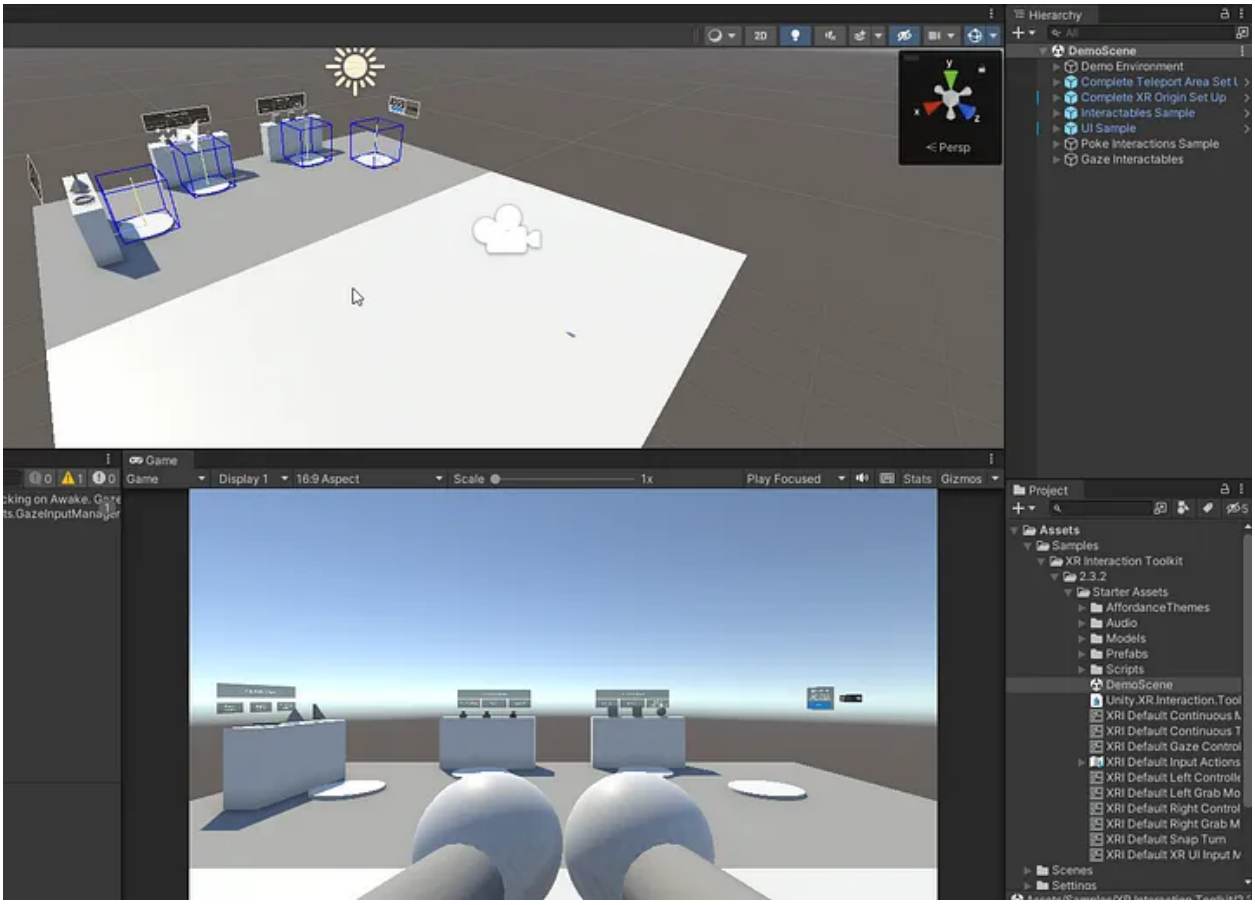

Figure 2.5: An example of a Unity 3D scene used for XR application development [34].

small variations in user position or motion can yield inconsistent outcomes. Automated frameworks must therefore simulate user movement, gaze, and controller events with high temporal accuracy to reproduce interactions reliably. Furthermore, evaluating user experience metrics such as comfort, presence, and immersion is inherently subjective and difficult to quantify automatically. Test scripts must also account for latency, rendering performance, and multi-modal feedback synchronization factors critical to avoiding motion sickness and preserving immersion. These complexities make XR testing an interdisciplinary task involving software engineering, human-computer interaction, and system performance analysis.

2.3 Generative AI

Generative AI refers to a class of artificial intelligence models designed to create new content, such as text, images, music, and even code. Unlike traditional AI systems that focus primarily on analysis or classification, generative AI systems learn patterns from very large datasets and use that knowledge to generate original outputs [38]. This capability has enabled a wide range of applications, from generating realistic human faces

to composing music or writing stories. As these models continue to improve, they are reshaping creative industries, education, software development, and communication by automating and enhancing tasks that once required human creativity [36, 37].

At the core of generative AI are machine learning models particularly deep learning architectures such as Generative Adversarial Networks (GANs) [80], Variational Autoencoders (VAEs) [81], and Transformer-based models [82]. GANs consist of two neural networks—a generator that creates synthetic data and a discriminator that evaluates its authenticity—trained in opposition to improve output realism. VAEs, on the other hand, learn a compressed latent representation of data and generate new samples by decoding points from this latent space. Transformer-based models, such as GPT, use a sequence-to-sequence attention mechanism that allows them to focus on relevant parts of the input when generating output. This self-attention mechanism enables Transformers to capture long-range dependencies in data more effectively than older architectures like recurrent neural networks (RNNs) or convolutional neural networks (CNNs) [39].

Training these models involves feeding them very large, diverse datasets and adjusting their internal parameters to minimize an error function that measures the difference between the model's predictions and the actual data. For example, in a text generation model, the training process uses millions or billions of sentences to teach the model to predict the next token (word or subword) given the preceding context. Over time, the model learns statistical patterns in grammar, vocabulary, and semantic relationships. This process requires high-performance computing resources, such as GPUs or TPUs, due to the vast number of parameters (often in the billions) and the complexity of the optimization process. Fine-tuning can then adapt a pre-trained model to specific domains or tasks, ensuring outputs are contextually relevant and aligned with user intent [83].

One of the examples of generative AI is ChatGPT, a large language model developed by OpenAI [35]. ChatGPT uses the Transformer architecture, specifically the decoder-only variant, to understand and generate human-like text based on input prompts. Its training involved two main stages: unsupervised pre-training on a massive text corpus to learn general language patterns, and fine-tuning using human feedback (Reinforcement Learning from Human Feedback) to improve output quality, helpfulness, and alignment with user expectations. The model's impact on daily life is increasingly visible, with applications in writing assistance, customer service, education, brainstorming, and even software development [35].

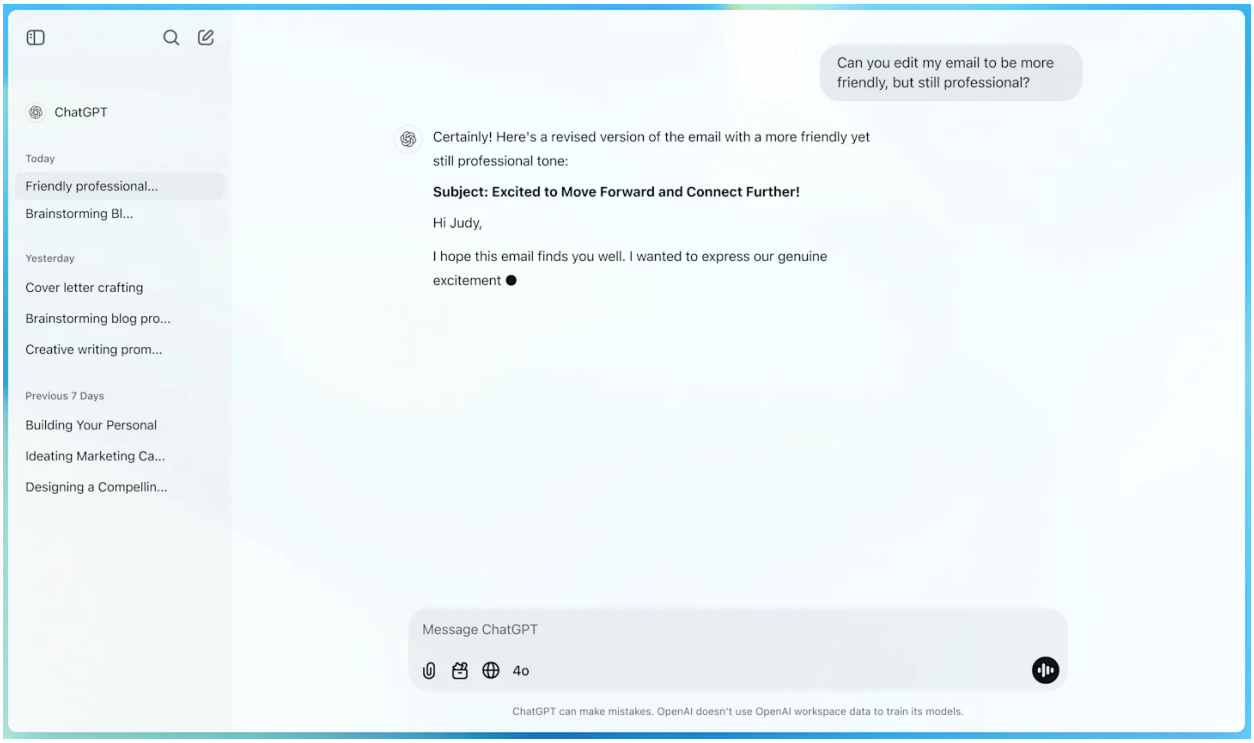

Figure 2.6: ChatGPT is actively generating a response based on the user's question. [35].

Chapter 3

Related work

This chapter provides an overview of the current research and studies related to our work, primarily focusing on software testing. It explores various concepts applicable to software testing and compares software methodologies across different application types.

3.1 Test Automation

Automated testing is an approach aiming to replace manual testing practices that are costly [8]. Several papers propose agent-based or rule-driven frameworks to enable scalable automation, making test automation practical for large projects. For instance, iv4XR and its extension [41, 42] use cognitive agents—autonomous software entities capable of reasoning, planning, and adapting their actions based on goals and environmental feedback to execute structured tests in dynamic environments, where system states and conditions can change in real time due to user interactions, AI behaviors, or procedural content generation. Similarly, AToVRS automates state-driven testing in Unity VR applications using simulated controller and headset input to mimic real user behavior [46]. These tools achieve high test coverage and are particularly suited for XR's continuous interaction space.

Model-based approaches have also proven effective in XR automation. The TestU framework [47] and Testing Virtual Reality Systems Using Model-Based Testing [48] both use state models or event-driven representations to automate GUI testing in Unity, capturing and replaying user interactions to identify regressions or inconsistencies. In addition, other solutions, such as TESTAR [43], reduce the need for manual scripting by exploring environments using a state-based model.

Broader surveys such as those by Qin and Weaver [49], Pan et al. [50], and the mobile app testing review [51] highlight common challenges like input variability, GUI dynamism, and scene interaction modeling—the

process of representing and simulating how users and system components interact within a virtual or physical scene for the purpose of testing. Capture-replay tools such as KORAT [52] and ACRT [53] demonstrate that automation is also applicable to embedded systems and Android applications, respectively, using external hardware or UI event interception. Complex real-time interactions refer to scenarios where multiple inputs, system responses, and environmental changes occur simultaneously and must be processed instantly to maintain functionality [52, 47]. Immersive interactions involve engaging the user through multisensory experiences, such as 3D visuals, spatial audio, and haptic feedback, where testing must account for both functional correctness and user experience quality [46]. These studies emphasize the importance of developing test automation tools that are platform-independent, minimally invasive, and capable of handling the complexity of real-time or immersive interactions, making automation a foundational requirement in modern software QA.

3.2 AI for Software Testing

Recent research has explored the use of AI-driven agents to automate software testing in XR environments. The iv4XR framework introduces a Java-based system built on the Belief-Desire-Intention (BDI) model, where agents perform structured, goal-driven testing instead of relying on random input generation. These agents use planning, navigation with A*, and a World Object Model abstraction to interact with XR systems. The framework demonstrates a modular and reusable approach to testing complex 3D applications such as Lab Recruits [41]. An extension of this work proposes cognitive agents capable of simulating user personas and assessing user experience using emotional models like FAtiMA [42].

Building on agent-based testing, scriptless test automation has been proposed to further reduce maintenance costs in XR environments. Pastor Ricós adapts the TESTAR model-based testing framework with the iv4XR [41] plugin, enabling automated interaction, navigation, and state validation without manual scripting. The tool supports the abstraction of XR environments and the verification of scene consistency, the process of ensuring that objects, layouts, and interactions in a scene remain as intended after changes or updates, using visual validation techniques, which involve comparing rendered images or frames against reference visuals to detect anomalies such as missing objects, incorrect positions, or rendering errors. This approach offers a scalable alternative to traditional scripted testing methods [43].

In addition, recent studies have investigated the use of Generative AI and Large Language Models (LLMs) for VR testing and development. Qin and Weaver evaluate GPT-4o [35] for black-box testing via field-of-view (FOV) analysis, showing moderate success in identifying objects and generating test actions, though spatial inconsistencies remain a challenge [44]. Separately, LLMs like ChatGPT have shown value in supporting

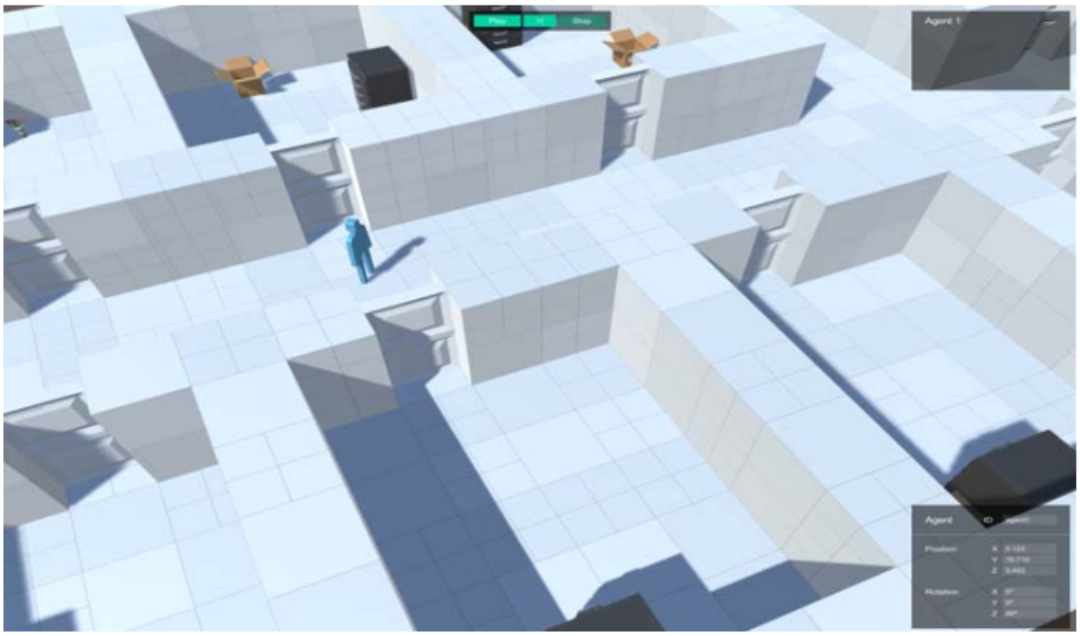

Figure 3.1: A 3D game is tested using the iv4XR framework, where agents verify game functionalities such as opening a door and entering a new room [41].

Unity-based XR development tasks such as scripting and debugging, particularly for novice developers. However, the effectiveness depends on precise prompting and tool-specific knowledge, highlighting current limitations and the need for deeper integration with XR development environments [45].

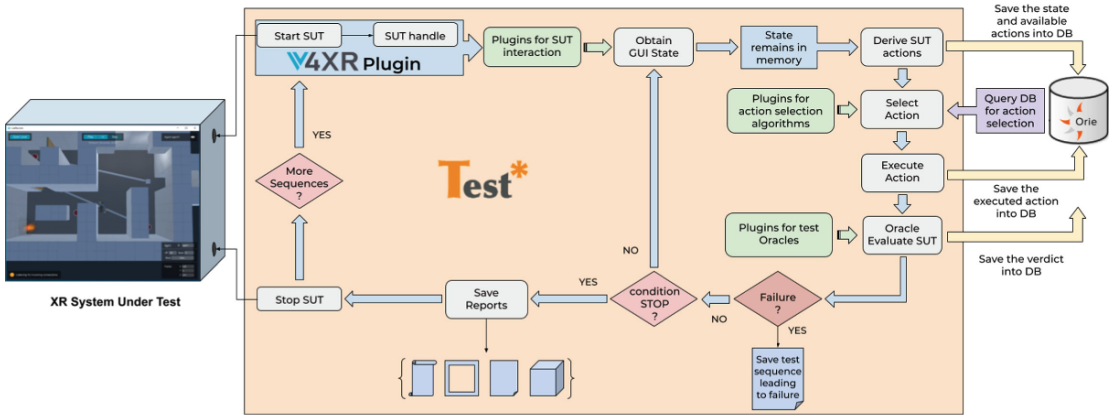

Figure 3.2: TESTAR operational flow [43].

3.3 Generatinve AI and Code Generation

Generative AI in software development enables developers to generate, complete, and refine code using natural language instructions. Tools like GitHub Copilot [62] and ChatGPT [35] leverage large language models, particularly transformer-based architectures, to understand and synthesize code across a wide range of programming tasks. As detailed by Donvir et al. [58], these tools support end-to-end development activities, including suggesting fixes, managing dependencies, and integrating CI/CD workflows. Ebert and Louridas [59] further note that such AI assistants enhance developer productivity by automating repetitive tasks, enabling engineers to focus on higher-level logic and design. Additionally, Nie et al. [61] highlight the conversational aspect of programming, showing that prompt engineering and iterative refinement have become integral to the modern development workflow. This collaboration between humans and AI is transforming programming into a more dynamic, interactive, and evolving process.

Despite these advancements, generative AI presents significant technical and ethical challenges. These include the risk of syntactically correct but semantically flawed code, intellectual property violations due to model training on public repositories, and over-dependence on AI systems by developers. Ebert and Louridas [59] emphasize the need for human oversight and validation in AI-assisted development, while Donvir et al. [58] highlight limitations related to cost, explainability, and lack of domain specialization. The Algorithms review [60] provides a broader context, noting the evolution of diffusion models, GANs, and autoencoders into powerful general-purpose generators. However, it also cautions about societal risks such as misinformation and skill erosion. Collectively, these studies underscore the importance of careful deployment of generative AI, guided by robust governance and ethical safeguards, as it offers remarkable capabilities for accelerating code generation.

Studies have shown that ChatGPT [35] is used across more than ten programming languages—most prominently Python and JavaScript for tasks ranging from debugging and interview preparation to academic assignments [85]. Empirical evaluations further indicate that its performance in both text-to-code and code-to-code generation tasks is highly dependent on prompt design [84]. These findings suggest that, while ChatGPT and similar models have potential for transforming software engineering, their effective integration into development workflows requires careful prompt engineering, consideration of user perceptions, and continuous evaluation of generated code quality [85].

3.4 Capture and Replay

Capture-and-replay is a software testing methodology in which user interactions such as inputs, navigation steps, and commands are recorded during a live session and later replayed to reproduce the same behavior automatically. This approach helps ensure consistent test execution, facilitates regression testing, and reduces the need for repetitive manual effort. Tools like ACRT [53] and KORAT [52] demonstrate early implementations in Android and embedded systems, where user inputs are recorded and later replayed to simulate behavior for testing. These systems are especially effective in restricted environments, where internal access to the system under test is limited. Frameworks like VRTest [54] extract information from a VR scene and control the user camera to explore the scene and interact with virtual objects.

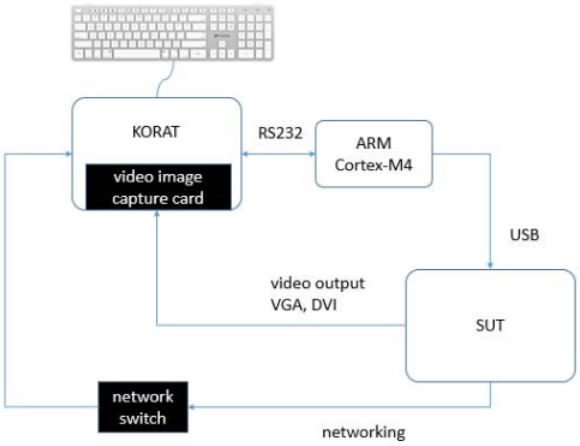

Figure 3.3: KORAT Capture/Replay architecture with keyboard [52].

In XR development, capture and replay also support automated test case generation and exploratory testing. The TESTAR framework [43], adapted for XR using the iv4XR plugin [41], combines scripted and scriptless automation with logic-based abstractions of recorded interactions. Meanwhile, tools like H5GF [55] provide a hybrid solution, allowing users to either record gameplay or write test scripts manually, which are then synchronized with the game engine for deterministic testing a process in which the same inputs and conditions consistently produce the same outputs, ensuring that test results are predictable and reproducible.

Beyond software testing, capture and replay are used for behavioral logging, user research, and training simulations. The PLUME framework [25] records and replays both behavioral and physiological data such as motion, gaze, and biometric signals collected during Unity based XR experiments. This enables researchers to analyze user behavior, share datasets, and replicate study conditions. Similarly, VRGym [56] uses recorded demonstrations to train AI agents in full-body interactions, supporting research in reinforcement learning and embodied AI. In both cases, replay serves as a foundation for reproducibility, model training, and

multimodal data analysis.

Capture and replay also extend to creative applications, such as content authoring and interaction modeling. In the Dialogues For One system [57], solo creators can record themselves multiple times to simulate multi-character VR scenes, enabling low cost dialogue creation and natural interaction flow.

Figure 3.4: Capture and Replay for Dialogue Creation [57].

Although capture-and-replay frameworks demonstrate value in automating repetitive testing, user behavior analysis, and creative prototyping, the current body of work reveals several limitations. Existing solutions often rely on manual setup, scene-specific instrumentation, and fixed interaction mappings that restrict scalability across diverse XR environments. Few systems effectively integrate semantic understanding of 3D scenes or leverage generative models to produce adaptive and reusable test cases. Furthermore, most frameworks are constrained to research prototypes or domain-specific tools rather than end-to-end, production-ready solutions capable of handling evolving Unity projects.

To address these gaps, this thesis introduces *TestGPT* a generative AI–driven testing tool designed to automatically create test scripts from Unity scene data and natural-language scenarios. By combining language-based reasoning with structured scene information, *TestGPT* bridges the divide between manual scripting and automated capture.

Chapter 4

Study Design

In this chapter, we present the study conducted to address the central research question: How do different testing approaches—manual coding, capture-and-replay, and the use of LLMs compare when applied to XR application testing? The primary objective is to compare these methods to understand their relative strengths, limitations, and practical challenges. A secondary objective is to evaluate the performance and usability of our custom-built tool based on generative AI. The chapter begins by outlining the study's structure, followed by a discussion of the participants, the pilot study, and an overview of the tasks assigned to participants.

4.1 Study Structure

The study was designed to evaluate and compare three different software testing methodologies, manual coding, capture-and-replay, and generative AI-based test generation using TestGPT. Each participant was asked to complete tasks using all three methodologies in a randomized order to minimize learning bias. The order of tasks was the same, but the order of testing methodologies was random. The goal was to assess each method's usability, efficiency, and perceived workload. At the beginning of the session, participants were asked to fill out a pre-study questionnaire and informed consent form. The questionnaire gathered information about their experience with Unity, XR development, test automation frameworks (such as Arium), and their familiarity with capture-and-replay tools and large language models (LLMs) for code generation. Copy of this forms is included in the appendix A.

For each of the three testing methodologies, the study followed a consistent structure. At the start of each stage, participants were given a brief introduction to the tool or technique relevant to that methodology. This included a short explanation and a live demonstration by researcher of a simple test scenario unrelated

to the actual tasks. This approach was informed by feedback from the pilot study, where participants requested hands-on examples to better understand each tool before being asked to use it independently. Following the demonstration, participants were given time to ask any questions. This preparation phase lasted approximately 10 minutes.

After the introduction, participants were given 20 minutes to complete the testing tasks using the respective methodology. During this time, they worked directly with a Unity project to write or generate test cases based on predefined test scenarios. At the end of each stage, participants were asked to complete two post-task questionnaires: the NASA Task Load Index (NASA-TLX) to assess perceived workload and the System Usability Scale (SUS) to measure tool usability. Finally, upon completing all three testing stages, participants took part in a short interview to provide qualitative feedback. They were asked about their experiences with each methodology, including any difficulties, limitations, perceived strengths, or suggestions for improvement. The number of completed test scenarios and any errors or issues encountered during the tasks were also recorded to aid in quantitative analysis.

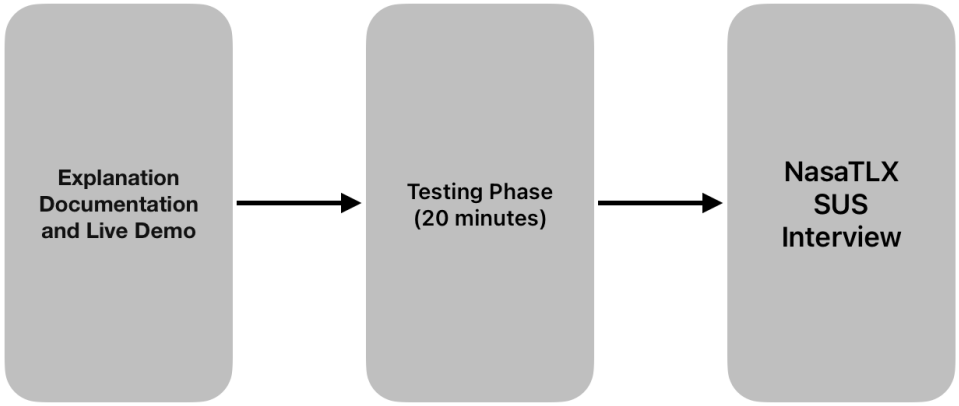

Figure 4.1: Structure of the Study

The study was conducted using a custom Unity project specifically designed for testing purposes. This sample project included several interactive GameObjects and UI elements, such as buttons that triggered specific actions (e.g., changing object colors or moving objects). These predefined behaviors served as the basis for the test scenarios assigned to participants across all three methodologies. All testing activities were performed on a Mac Mini M2 system, which was used consistently throughout the study to ensure a controlled and uniform testing environment. Participants interacted with the Unity Editor directly on this device, using it to view the project, execute test scenarios, and validate outcomes. This setup allowed for reliable observation and data collection across different testing strategies under the same technical conditions.

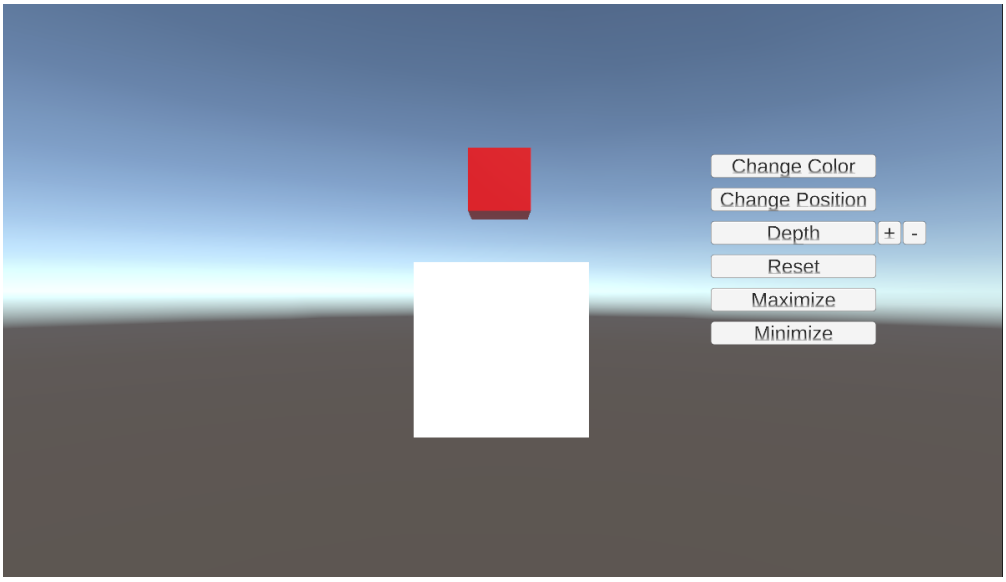

Figure 4.2: Test Scene.

4.2 Pilot Study

Before conducting the main user study, we carried out a pilot study with three participants. The purpose of the pilot was to verify that the overall process functioned smoothly and to identify any usability issues or sources of confusion in the documentation for each phase. It also provided early feedback on how participants interpreted the instructions and interacted with the tools and tasks. By simulating the entire workflow with a small group, we were able to detect and address potential problems that could compromise the reliability, clarity, or consistency of the actual study setup.

One key issue that emerged during the pilot study was related to the documentation provided to participants. Initially, we provided written instructions for each stage of the study and expected users to follow them independently. However, participants reported that the documentation was sometimes unclear, particularly when it involved tools or frameworks they were unfamiliar with. This was because some tools were new to the users, and reading about them and applying that knowledge immediately proved difficult. Based on this feedback, we decided to include a live demonstration of an example test scenario before participants began the main study tasks.

Additional feedback revealed that participants benefited from having time to explore the Unity project and interact with objects prior to completing the tasks. This hands-on exploration allowed them to become familiar with the functionality and layout of the application they were testing. Lastly, participants noted that for the manual coding tasks, it was helpful to have access to the implementation files containing the logic of the features being tested. Viewing the source code gave them a clearer understanding of what needed

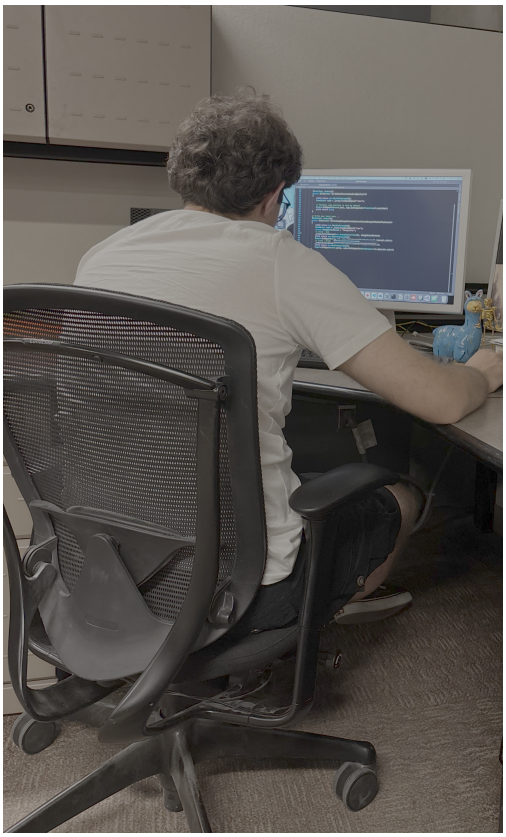

Figure 4.3: A study participant, working with the Unity editor and designing a project, performed tasks using a Mac Mini for this study.

to be validated and how to write the corresponding tests. These insights led us to refine the study design by incorporating introductory demonstrations, pre-task exploration time, and access to relevant source code. As a result, the main study ran without any of the confusion encountered during the pilot.

4.3 Participants

To recruit participants for the study, we sent a call for volunteers to graduate students and developers within the ICT department, specifically targeting individuals with some experience in Unity or XR development. One of the primary eligibility criteria was a minimum level of familiarity with developing XR or 3D applications in Unity, as this study focused on evaluating different software testing methodologies for such environments. In total, 15 participants took part in the study, including 10 men and 5 women. All participants were volunteers and provided informed consent before the study began.

As part of the pre-study questionnaire, we collected data on participants' backgrounds and technical experience to better understand their skill levels and how these might influence the study's results. One key

parameter was their experience with XR development in Unity, as this was directly relevant to the testing scenarios they would encounter. Since the research involved validating the functionality of XR applications using Unity, it was crucial to ensure that participants were at least familiar with working within the Unity environment, including writing scripts and navigating scenes.

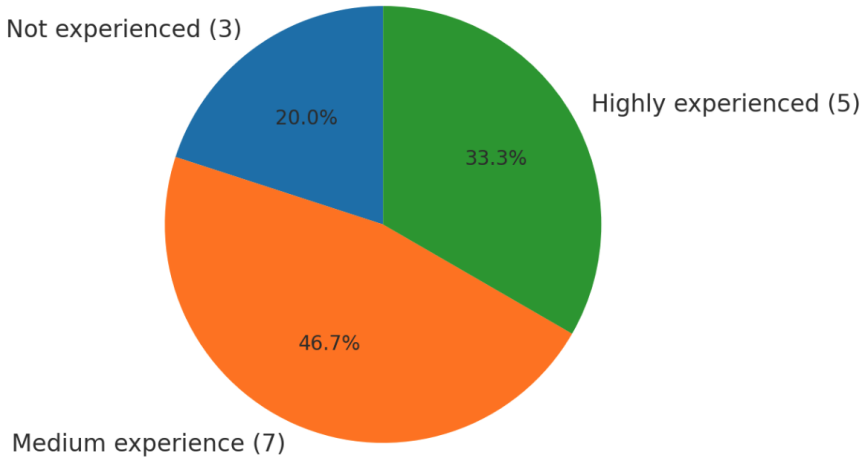

Figure 4.4: Experience of participants in the development of XR or 3D applications in Unity.

Another important factor we investigated was participants' familiarity with software testing practices. Since the study focused on evaluating three different testing methodologies, manual testing, capture-and-replay, and generative AI-based test generation, we expected that individuals with greater testing experience might complete tasks more efficiently or recognize issues more easily. To distinguish between these skill levels, participants were asked to self-report their software testing experience, allowing us to analyze whether prior knowledge had an impact on their performance.

Finally, we asked about participants' familiarity with three specific tools or techniques central to the study. These included the Arium framework (used for manual and scripted test writing), generative AI for code generation (relevant to using TestGPT), and capture-and-replay testing tools, like Selenium or GameDriver. Participants were asked whether they had prior experience with each, so we could differentiate between users encountering a tool for the first time and those already familiar with it. This information allowed us to interpret participant performance more accurately and understand how tool familiarity influenced test effectiveness and usability perceptions.

Among all the participants, three had prior experience with the capture and replay testing method, all of whom used Selenium, a capture and replay testing tool for web applications. No one had experience with this software testing methodology for Unity applications.

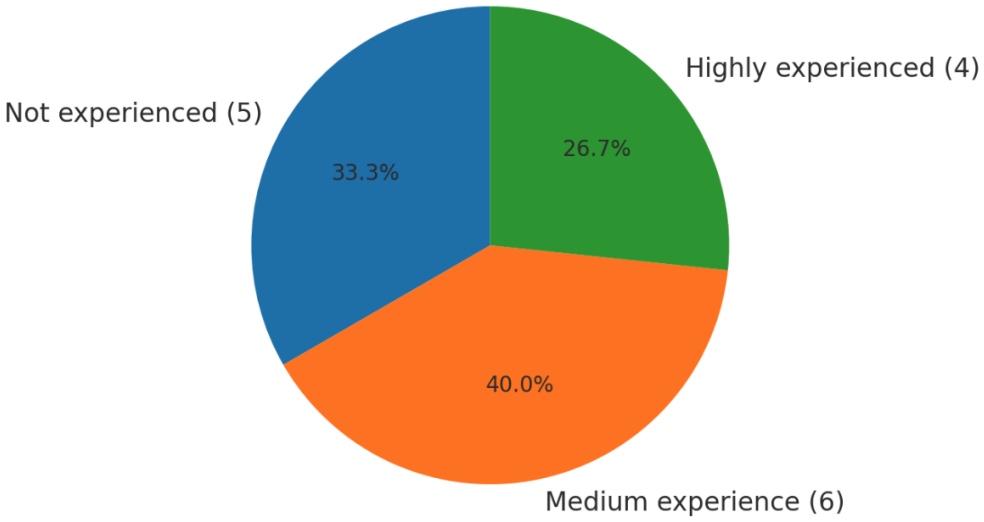

Figure 4.5: Experience of participants in software testing.

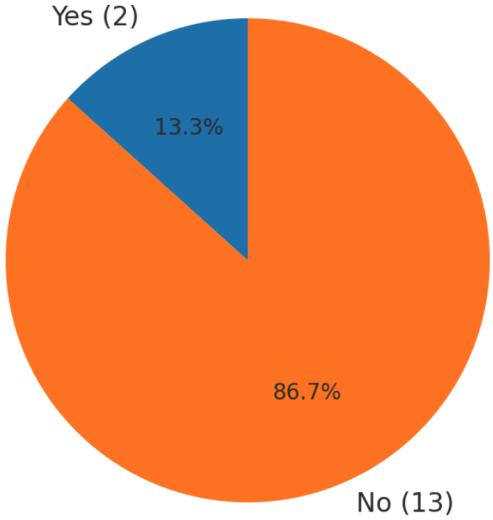

Figure 4.6: Participants' experiences with the Arium Testing Framework.

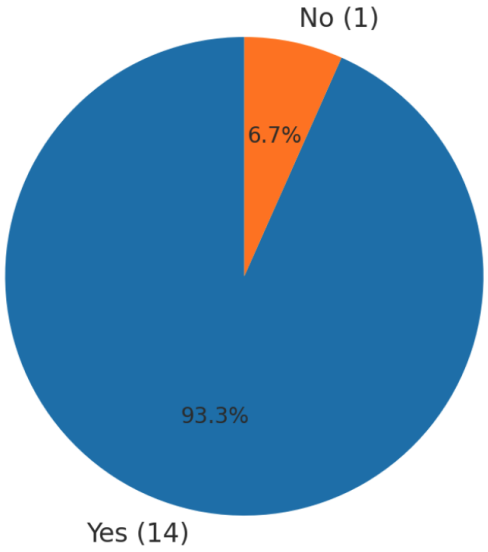

Figure 4.7: Participants' experiences with code generation using generative AI.

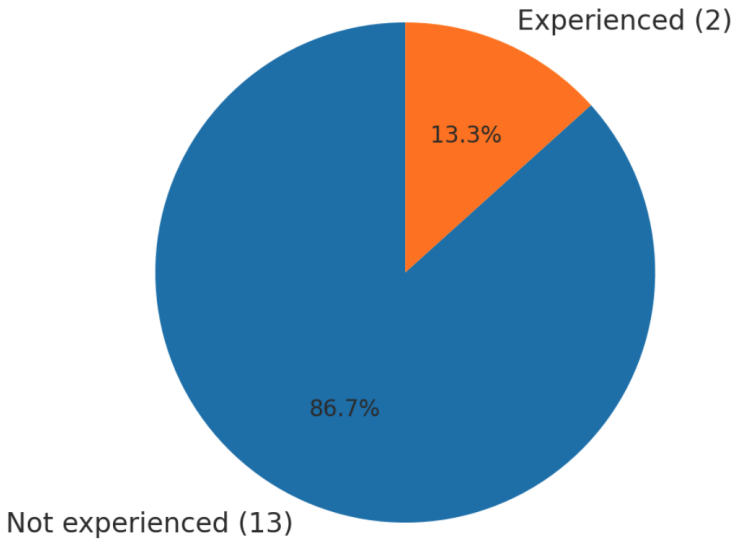

Figure 4.8: Participants' experiences with the capture and replay testing methodology.

4.4 Tasks

To support the evaluation of different software testing methodologies, we developed a custom Unity project that served as the foundation for all participant testing activities. The scene was designed with simplicity and clarity in mind and featured two cubes, a red cube and a white cube, along with six interactive buttons, each responsible for modifying a specific attribute of the white cube. The red cube served as a visual reference and remained unchanged throughout the study. We used it to observe how *TestGPT* could understand and distinguish between objects in a scene using only a high-level description of the test scenario and the Unity scene file. All interactive behaviors were implemented in a Unity script file named `ChangeScript.cs`, which was accessible to participants during the manual coding phase. The interactive buttons provided in the Unity scene were as follows:

1. **Change Color** – changes the color of the white cube to red and green.
2. **Change Position** – updates the cube's position in the 3D space.
3. **Increase Depth** – increases the z-axis depth of the cube up to a predefined maximum.
4. **Decrease Depth** – decreases the z-axis depth of the cube down to a predefined minimum.
5. **Reset** – resets the cube to its default color, position, depth, and scale.
6. **Maximize / Minimize** – scales the cube up or down, modifying its overall size.

Each of these buttons triggered a specific function implemented within `ChangeScript.cs`. However, some functions were intentionally designed with small bugs to evaluate whether participants using different testing approaches could identify and report these issues effectively.

The first bug was in the `Change Color` function, where the color cycle was incorrect—the sequence skipped the red color once in each cycle, even though the function was supposed to switch consistently between red and green. The second bug occurred in the `Change Position` function, which should have moved the cube sequentially to the right, left, up, and down. Instead, the left position was skipped, which was considered a defect. The third bug was in the `Change Depth` function, where the behavior was inverted—selecting "increase depth" actually decreased the cube's depth in the scene. Participants needed to check the cube's Z-axis value to detect this issue. The final bug was in the `Minimize` function, where the scaling logic was incorrect, resulting in the cube not becoming smaller as intended.

Participants were provided with clear documentation outlining the expected behaviour of each functionality. They were then asked to validate whether the actual behaviour matched the documented expectations.

Any error should be reported as a fault or bug. The goal was to simulate a realistic testing scenario where developers or QA engineers must confirm that user-facing functionalities behave correctly across various interaction conditions. Participants were asked to test the following six scenarios:

1. **Color Change:** Verify that clicking the "Change Color" button updates the white cube's color to green and red.
2. **Position Change:** Test whether clicking the "Change Position" button correctly updates the cube's position.
3. **Depth Adjustment:** Validate both the "Increase Depth" and "Decrease Depth" buttons to ensure the cube's z-position is updated within allowed limits.
4. **Reset Functionality:** Confirm that the "Reset" button restores the cube's attributes (position, color, depth, and scale) to their default states.
5. **Maximize:** Test the functionality of the "Maximize" button to ensure the cube's scale increases appropriately.
6. **Minimize:** Check whether the "Minimize" button correctly reduces the cube's scale.

Although the Unity scene was intentionally simple, the tasks were selected to capture fundamental categories of interaction that frequently occur in real-world XR applications. Object manipulation, including translation, rotation, and scaling, underpins most XR workflows ranging from manipulating virtual tools in training simulations to adjusting components in industrial or architectural design environments. Color change represents visual state feedback, a common element in XR interfaces for conveying selection, status, or task completion. Depth adjustments correspond to spatial positioning and depth perception, both critical in testing scene rendering accuracy and user spatial awareness. Together, these tasks encompass visual, spatial, and interactive feedback loops central to immersive system design.

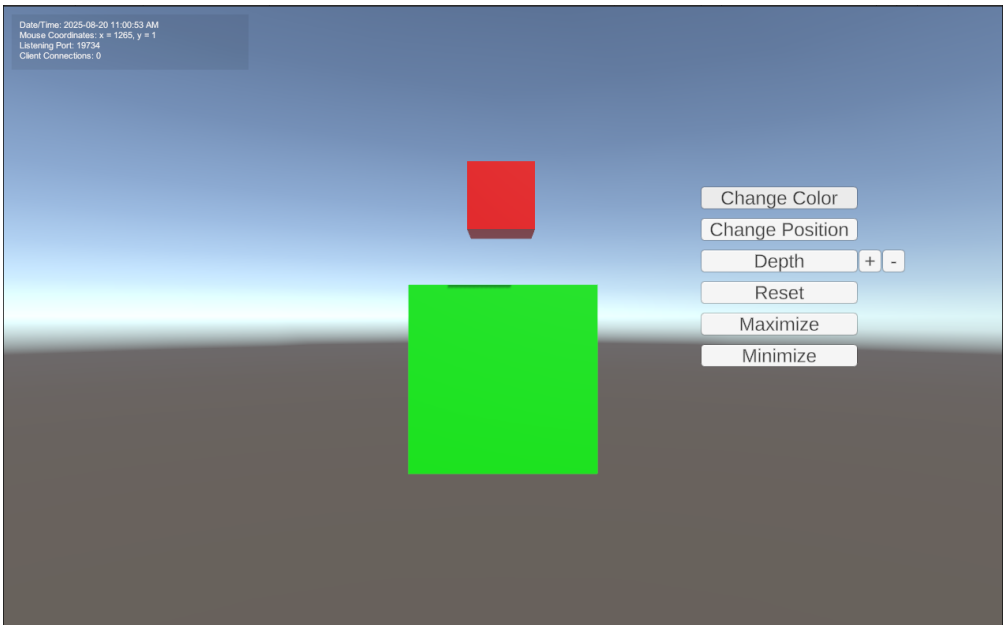

Figure 4.9: Change Color Functionality – The cube's color changed to green at first.

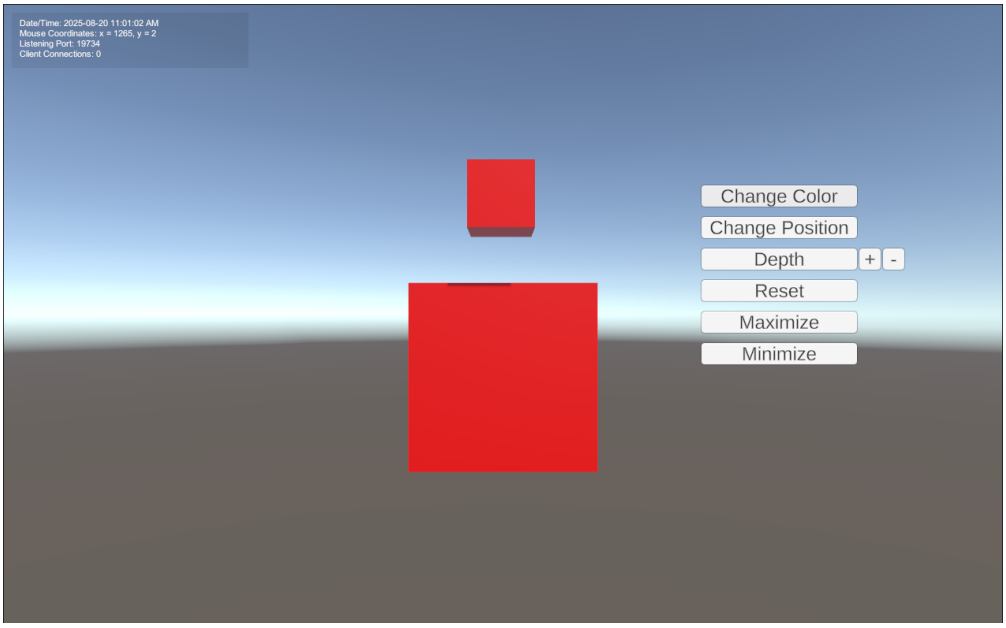

Figure 4.10: Change Color Functionality – The cube's color changed to red after green.

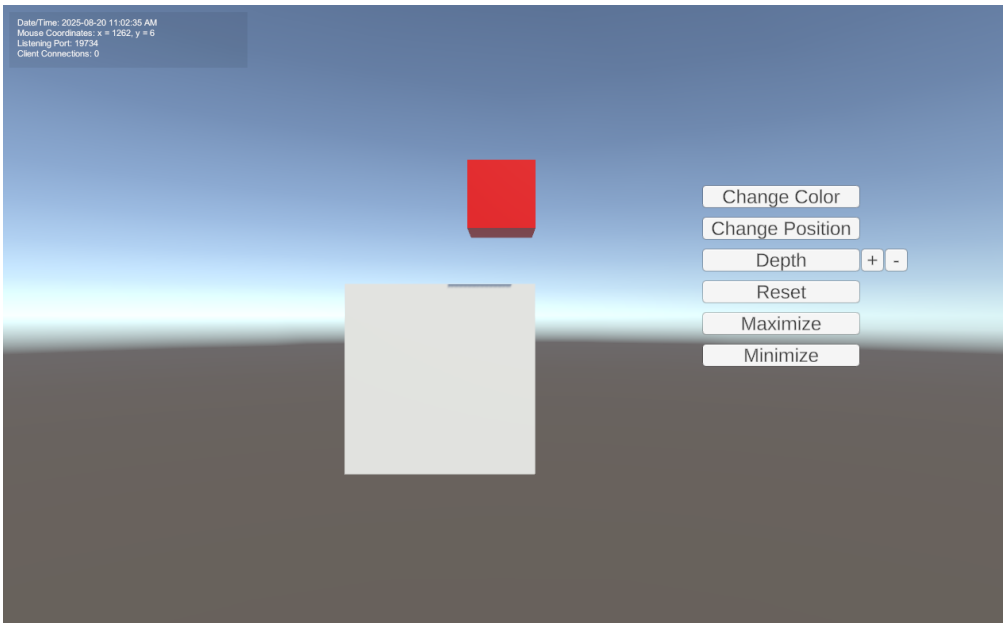

Figure 4.11: Change position Functionality

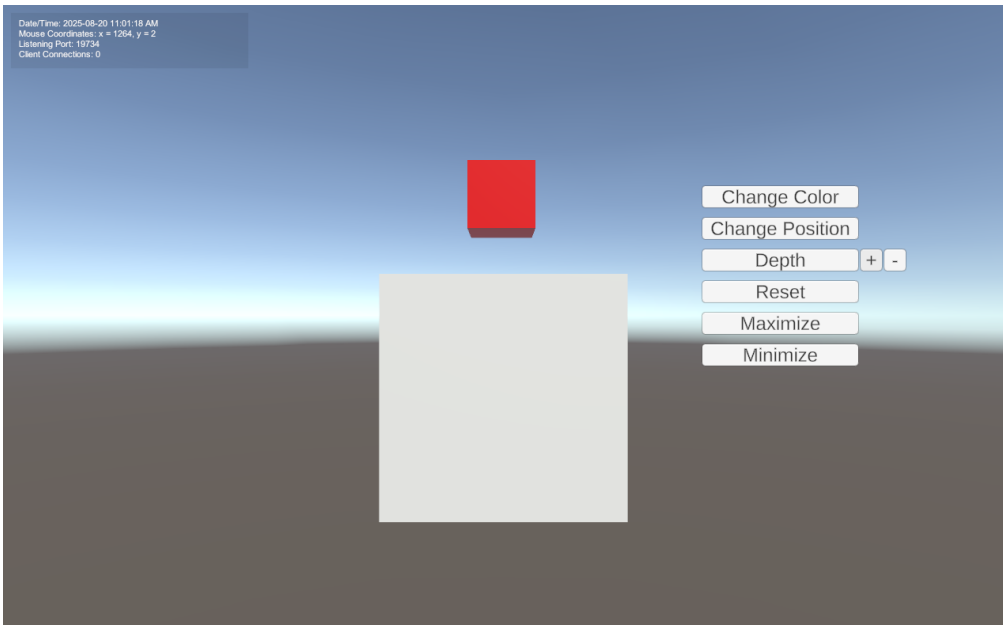

Figure 4.12: Change Depth Functionality – Decreasing the depth

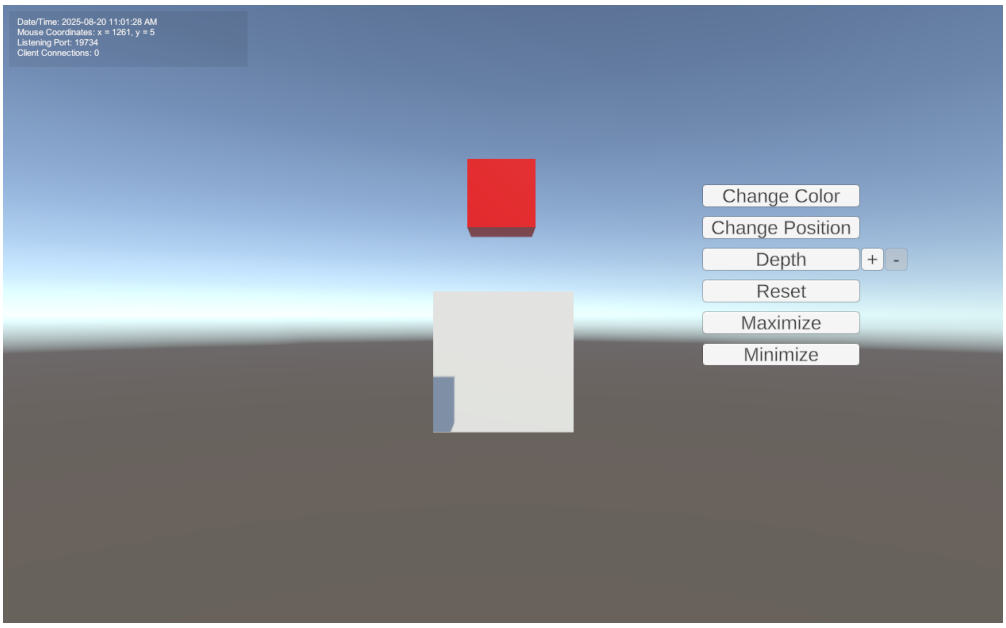

Figure 4.13: Change Depth Functionality – Increasing the depth

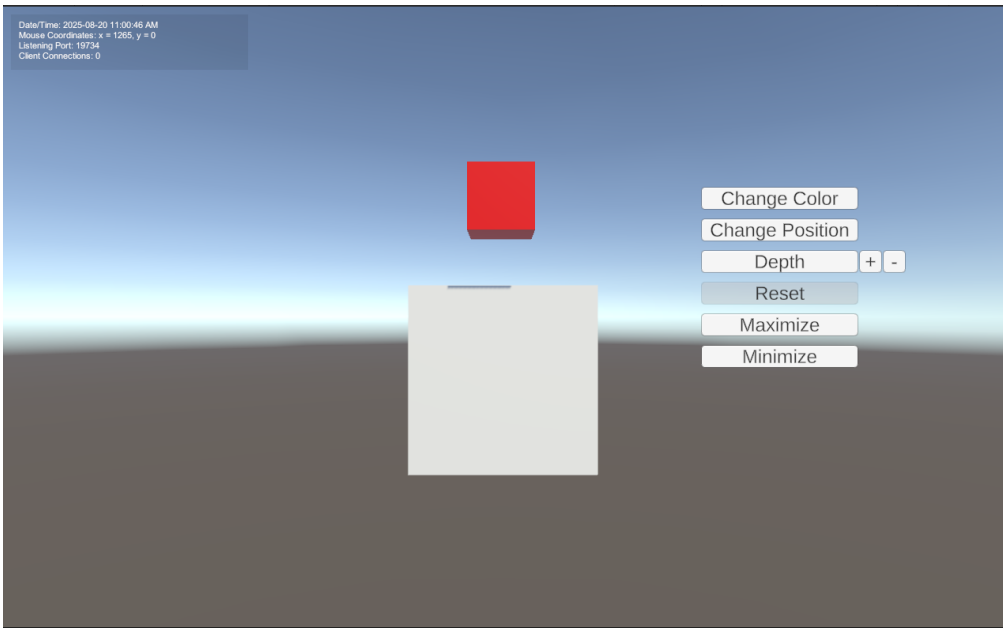

Figure 4.14: Reset Functionality – Change back all the attributes to their initial values.

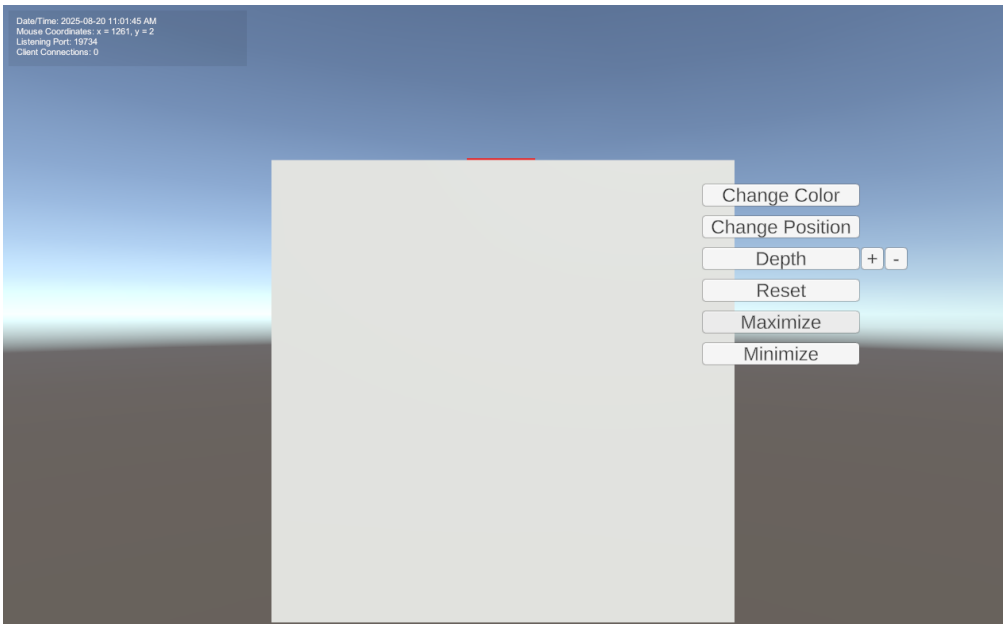

Figure 4.15: Maximize Functionality – Scales up the cube's size.

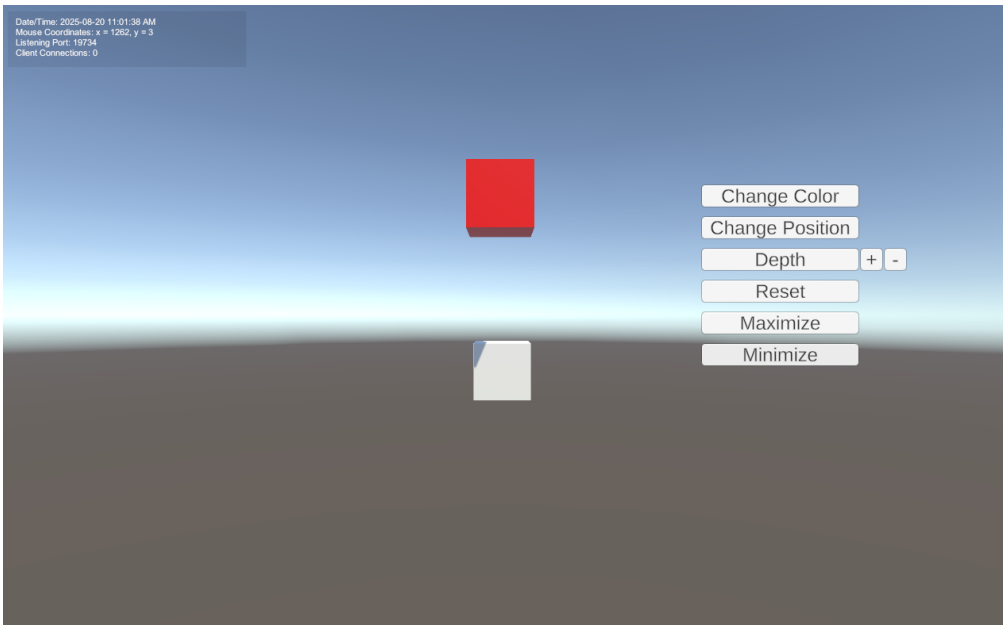

Figure 4.16: Minimize Functionality – Scales down the cube's size.

```
using System.Collections;
using AriumFramework;
using AriumFramework.Plugins.UnityCore.Extensions;
using AriumFramework.Plugins.UnityCore.Interactions;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.TestTools;

namespace Samples.AriumSample.Tests
{
    public class SampleSceneTests
    {
        private Arium _arium;

        [OneTimeSetUp]
        public void SetUp()
        {
            _arium = new Arium();
            SceneManager.LoadScene("SampleScene");
        }

        [UnityTest, Order(1)]
        public IEnumerator VerifyCubePositionIsZeroByDefault()
        {
            yield return new WaitForSeconds(2);
            GameObject cube = _arium.FindGameObject("Cube");

            // Validate cube position is zero by default
            Assert.AreEqual(Vector3.zero, cube.GetComponent<Transform>().position);
            yield return null;
        }

        // Write your tests here ...

    }
}
```

Figure 4.17: The Arium Base code, along with an example for participants to manually code, is provided to test the initial position of the white cube, referred to as Cube.

4.5 Ethics Considerations

This study was reviewed and approved by the Conjoint Faculties Research Ethics Board (CFREB) at the University of Calgary under Ethics ID **REB25-0214**. Prior to participation, each volunteer received a detailed consent form outlining the study's purpose, procedures, and their rights as participants. Written informed consent was obtained from all participants before data collection.

Participation in the study was entirely voluntary, and individuals were informed that they could withdraw at any time without penalty. To protect privacy, all collected data were anonymized, and no personally identifying information was stored or reported. Only aggregated results were analyzed and presented in this thesis. The research was assessed as involving minimal risk, focusing on participants' interaction with software testing tools rather than on sensitive personal data. Copies of the CFREB ethics certificate and the study consent form are included in the appendix for reference.

Chapter 5

System Design

In this chapter, we present the systems and tools used throughout this thesis. We begin by introducing TestGPT, the tool we developed based on generative AI, along with an explanation of what can be tested using it and how the tool operates. To clarify its functionality, we first provide an overview of the structure of a Unity scene and how it is stored. We then describe the Arium framework, the test automation framework on which our generative AI relies to generate test scripts. The same framework was also used for manually developing test scripts. Finally, we provide an overview of the tool selected for the capture-and-replay testing approach.

5.1 Unity Scene File

To design a tool based on generative AI for automatically generating tests for XR applications, it is first necessary to understand the structure of Unity scene files. Writing meaningful test scripts requires access to detailed information about the scene, since the TestGPT must rely on this data to generate tests. Understanding how Unity stores components within a scene is therefore a critical step in developing the tool proposed in this research. Because the projects under study are built in Unity, knowledge of how scenes and their data are represented is essential. For example, if a project contains several objects and we want to test whether their color can be changed to blue, the tool must be able to identify those objects in the scene and access their properties in order to apply the modification and verify its correctness.

In Unity, scene files are structured to represent a hierarchy of objects. Each component within the scene, such as GameObjects and cameras, is organized under a parent–child system. Unity stores all this information in a scene file, which is technically a YAML file [62]. This YAML file contains detailed data for every object in the scene, including their properties and attributes. Even default elements like cameras

are included in this structure. All the attributes related to each object—such as position, rotation, scale, material references, and scripts—are stored in this file, and Unity automatically updates the YAML files whenever any changes are made to the attributes of objects in a scene. This means we can access the most up-to-date object properties simply by parsing and analyzing the YAML file, and we can also make changes to the scene by modifying the values of object attributes in the file. Therefore, by parsing these files, we not only gain access to the information we need but also obtain the ability to apply modifications directly.

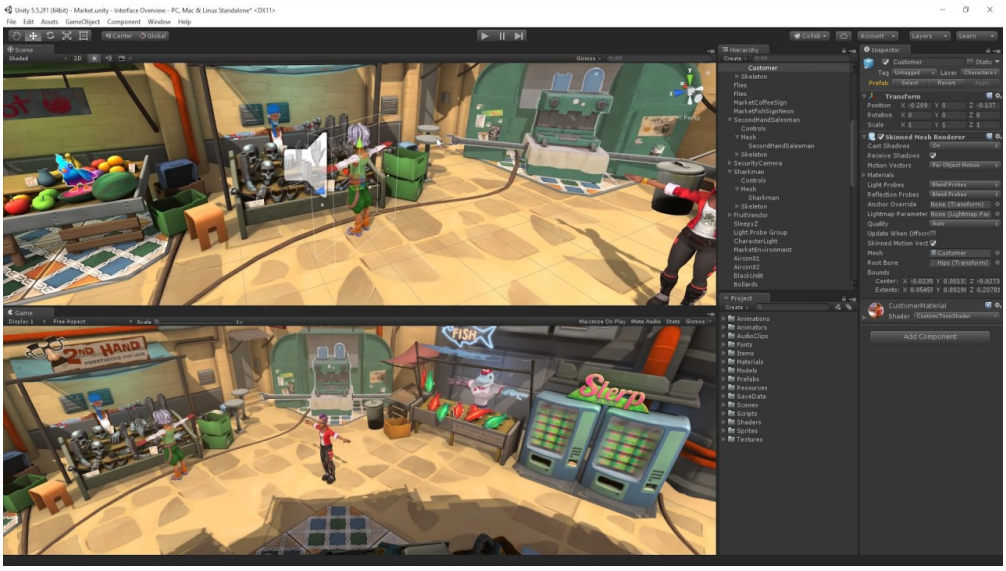

Figure 5.1: A Unity scene for a 3D application [64].

To modify an object's attribute, we need to identify it—either by its name or ID, and locate it within the YAML structure. Once the object is found, we can access and update any of its attributes. By doing so, we effectively modify the object inside the Unity scene without opening the Unity Editor. In fact, altering the object's data in the YAML file directly reflects changes in the actual scene.

In this example, we demonstrate this concept using a GameObject named "Cube" By examining the YAML file associated with the scene, we can locate the cube using its name or unique identifier. Once found, we can inspect and modify its attributes, such as position, rotation, and scale. This means that any changes made directly in the YAML file are equivalent to making the same modifications within the Unity Editor. This approach allows automated tools to interact with Unity scenes programmatically and provides a foundation for manipulating scene data externally.

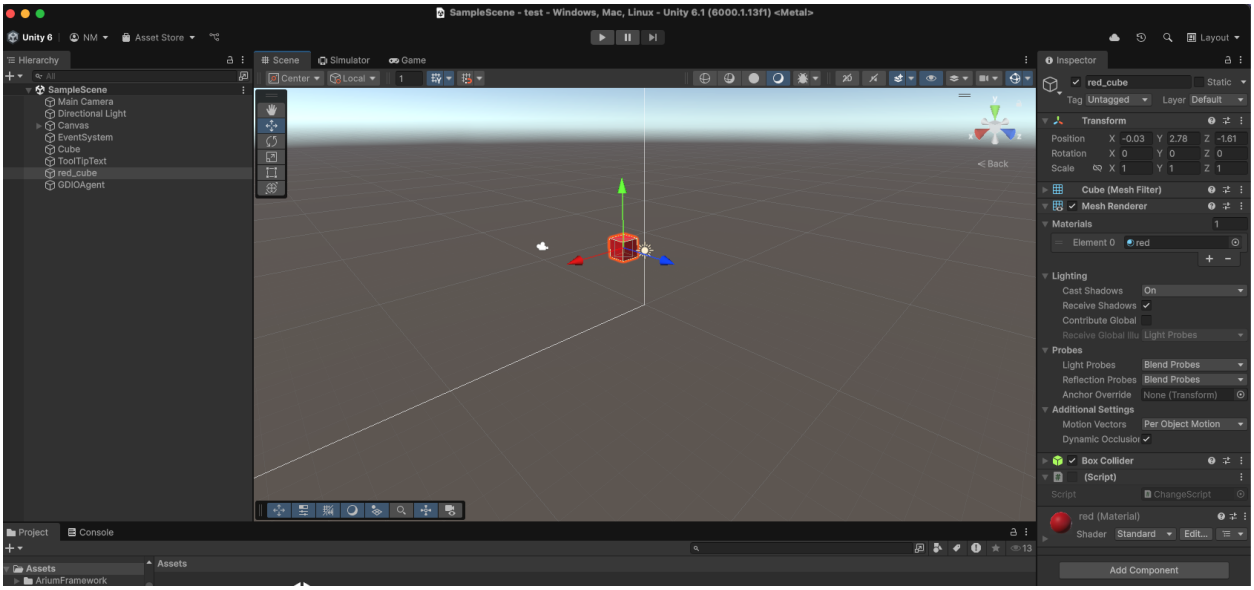

Figure 5.2: A Unity scene with an object called red_cube.

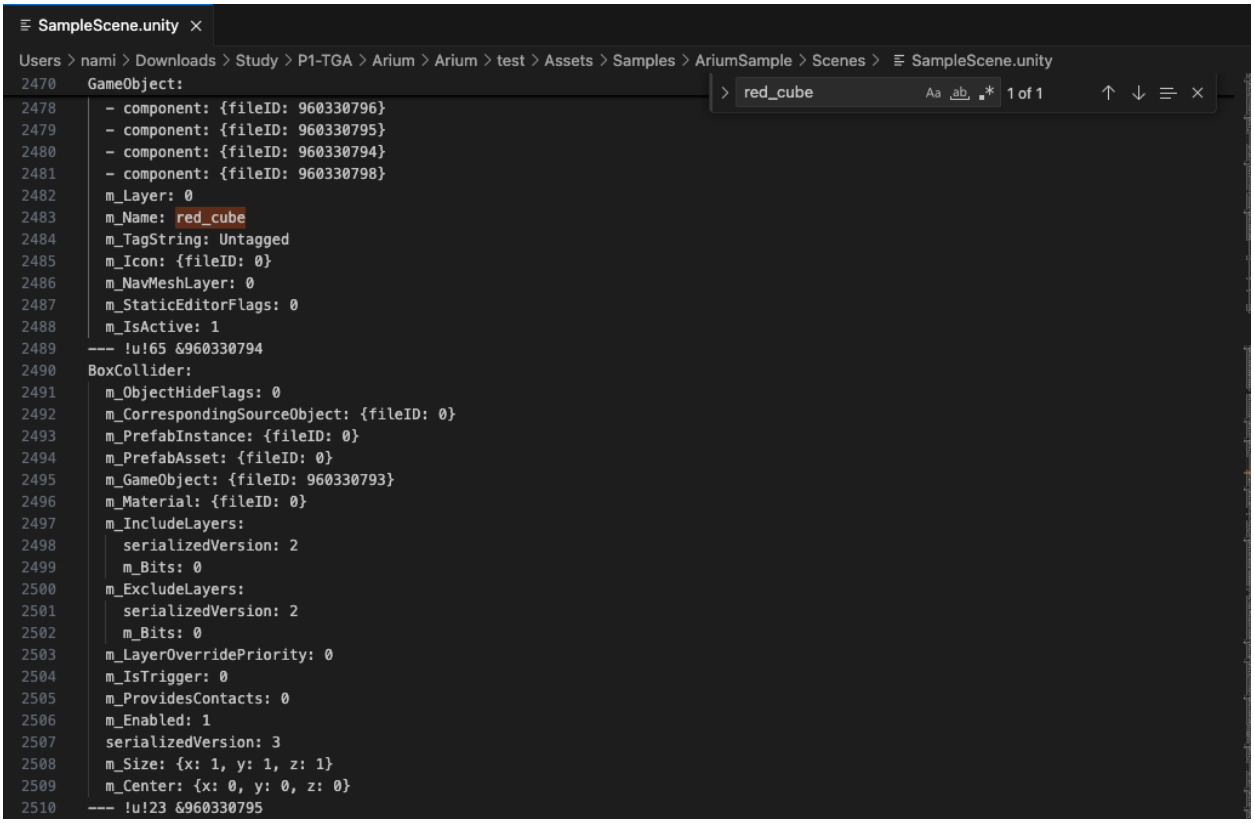

Figure 5.3: Finding the red_cube object in the YAML file.

5.2 Arium

Arium [24] is a functional test automation framework specifically designed for 3D and XR applications built with Unity. Arium operates within Unity's object hierarchy by identifying and interacting with GameObjects. This framework supports cross-platform testing—including the Unity Editor, Android, and Universal Windows Platform (UWP), enabling automated testing of user interactions in immersive environments. Arium allows testers to simulate runtime interactions such as pointer clicks, drag events, animation triggers, and physics-based behaviors using Unity's event interfaces like PointerClickHandler or DragHandler. These actions are written in C# using Unity's testing infrastructure and executed via the built-in Test Runner, allowing seamless integration with existing Unity development workflows [17, 63]. For example, in the following test code, we define a test function that checks the position of an object called Cube. Each test function must first be assigned an order number, since all tests are executed automatically in the order specified. In this function, we first retrieve the object named Cube and then use an assertion to verify its position.

```
[UnityTest, Order(3)]
public IEnumerator VerifyCubePositionIsZeroByDefault()
{
    yield return new WaitForSeconds(2);
    GameObject cube = _arium.FindGameObject("Cube");

    // Validate cube position is zero by default
    Assert.AreEqual(Vector3.zero, cube.GetComponent<Transform>().position);
    yield return null;
}
```

Figure 5.4: A sample test function based on the Arium test automation framework.

Arium simplifies the process of automating XR scenarios by offering intuitive methods such as FindGameObject() for locating elements in the scene hierarchy and PerformAction() for triggering specific behaviors. For example, a test can validate whether a UI text component updates correctly after a button click by programmatically simulating the event and asserting the resulting state which means checking the outcome of the event wit hte expected value. Developers integrate Arium into their Unity projects by importing the provided UnityPackage and creating a Tests folder to manage scripts.

5.3 TestGPT[1]

TestGPT is a generative AI-based tool developed as part of this research to streamline the creation of automated test scripts for Unity XR applications. Built on top of OpenAI's o3-mini-high model [65, 35],

[1]Available at: https://github.com/NamiMod/testgpt_deploy

TestGPT processes high-level test scenarios provided by users and generates executable test code using the Arium testing framework. The tool is specifically designed for Unity applications, leveraging Unity's YAML-based scene files to extract object hierarchies and attributes. This enables contextual understanding, meaning that TestGPT is not only aware of which objects exist in the scene but also how they are structured and what attributes they have.

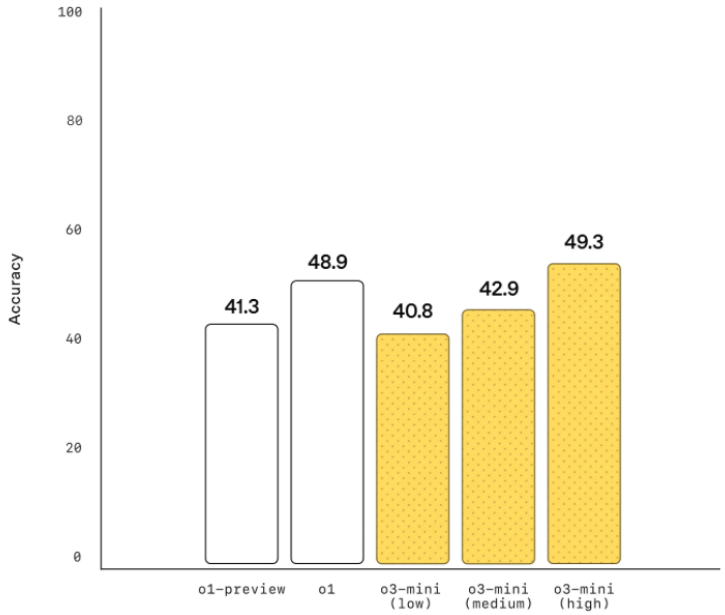

Figure 5.5: Figure 5.6: This figure compares the accuracy of OpenAI's o3-mini model to other models in the Software Engineering (SWE-bench Verified) benchmark. The numbers above each bar indicate the percentage of bug-fix tasks solved correctly by the model. For instance, o3-mini at high reasoning effort achieves an accuracy of 49.3%, meaning it successfully resolves nearly half of the benchmarked software bugs. While 50% may seem modest in general terms, in the context of automated program repair where tasks involve reasoning over large complex codebases, this performance is considered good [65, 86].

TestGPT leverages OpenAI's o3-mini, a lightweight generative-AI model with reasoning capabilities, to translate high-level test scenarios into executable C# test scripts for the Arium framework. The model is prompted with both the user's natural-language test scenario and structured context extracted from Unity's YAML scene files such as attributes. We also supply Arium's syntax and illustrative examples so that the generated code is syntactically valid for the Arium framework and logically consistent with the scene state. We selected o3-mini to assess the feasibility of generative AI for XR software testing; although other models involve different trade-offs, a comparative evaluation is outside the scope of this research.

The workflow of TestGPT is designed to be user-friendly. The user provides a high-level description of the desired test scenario without needing to specify technical details such as object names or identifiers and uploads the corresponding Unity scene file. TestGPT then analyzes the YAML data to identify relevant

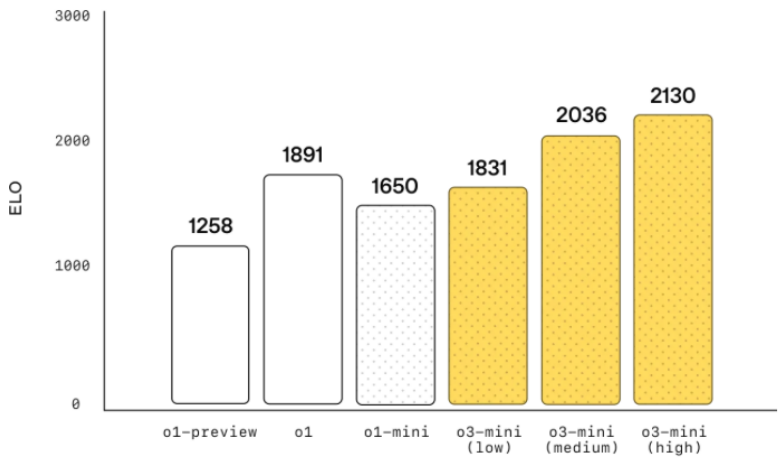

Figure 5.6: This figure compares the accuracy of OpenAI's o3-mini model to other models in the Competition Code (Codeforces). On Codeforces, OpenAI's o3-mini model achieves progressively higher Elo scores with increased reasoning effort, outperforming its predecessor, o1-mini. With medium reasoning effort, it matches o1-mini's performance [65].

GameObjects and their properties. The user has the freedom to include more than one object in the test scenario. If there are multiple similar objects in the scene, such as some cubes, TestGPT attempts to identify the correct one based on the additional descriptions provided by the user. Based on this analysis, the tool generates Arium-compatible test code, which can be copied into the Unity project's test folder and executed directly using the Unity Test Runner, a built-in Unity tool.

One of the features of TestGPT is its ability to simplify the description of test scenarios. For instance, in a case where a user wants to verify whether clicking a "Change Color" button updates the color of an object called cube, they only need to write: "Test if you can change the color of the cube by clicking on the change color button." Instead of writing a long and detailed description such as: "Test if you can change the color of an object in the scene called cube at the position (0,4,2) by performing a click action on another object called change color, which has a script attached for being clicked and is located at position (0,10,2)." TestGPT automatically handles the rest, locating the cube and the button in the scene file, extracting their identifiers and attributes, and generating code that performs the required interactions and assertions. This level of abstraction allows users to focus on the testing logic rather than implementation details.

TestGPT supports a variety of assertions, including equality assertions, inequality assertions, boolean assertions, null assertions, reference assertions, exception assertions, collection assertions, string assertions, and floating-point assertions.

TestGPT also supports more complex scenarios involving multiple objects and interactions. For example, if a test requires verifying that a white cube moves to the position of a blue cube when a "Change Position" button is clicked, the user only needs to describe the scenario at a high level. TestGPT analyzes the scene file

both spatially—by interpreting the positions and relationships of objects in 3D space—and semantically—by interpreting object names, attributes, and roles. By combining these two perspectives, the tool can identify the correct objects, infer their relationships, and generate the appropriate test steps automatically.

The use of generative AI in TestGPT also introduces flexibility in test specification. Users can describe multiple sequential interactions within a single prompt, and the model will produce comprehensive test logic to reflect that flow. This enables rapid prototyping and regression testing without the need for extensive manual scripting. Additionally, by relying on the Arium framework for execution, TestGPT ensures that generated tests are modular, reusable, and compatible with Unity's PlayMode test pipeline.

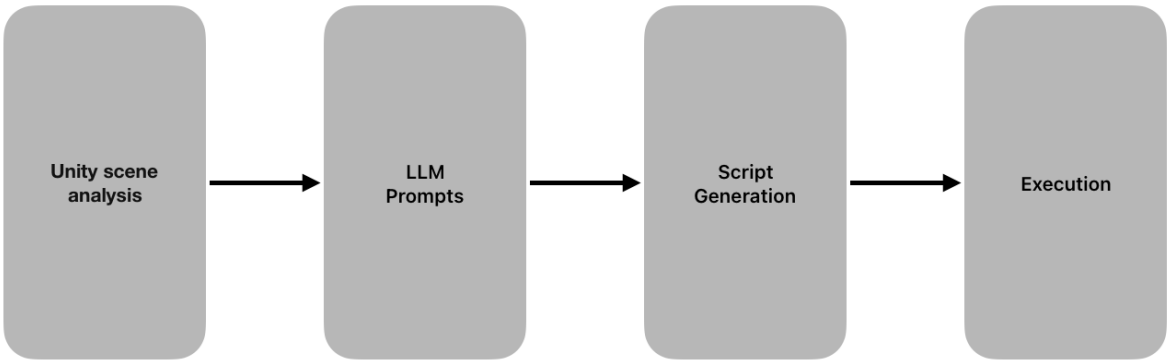

Figure 5.7: TestGPT's workflow

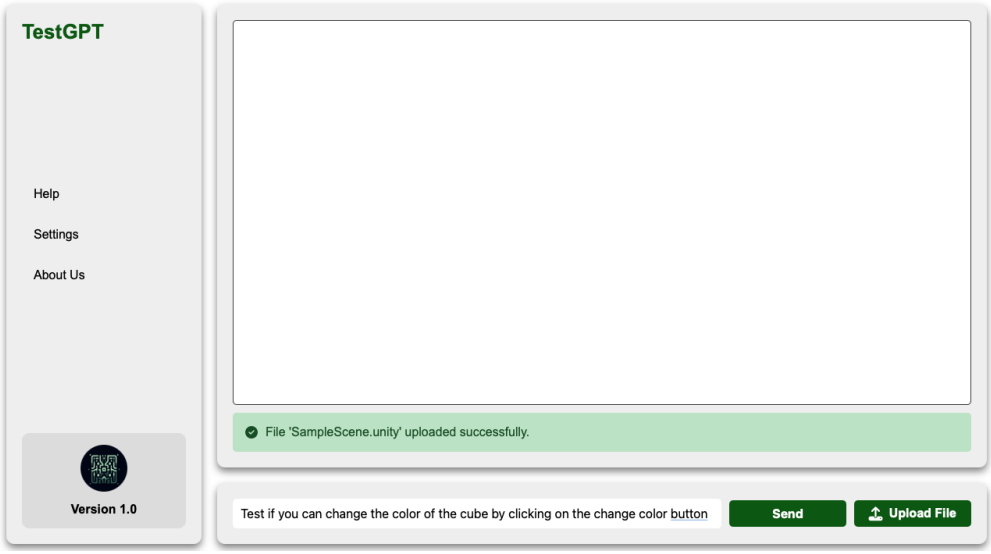

Figure 5.8: Testers must upload a scene file and write the test scenario in natural language.

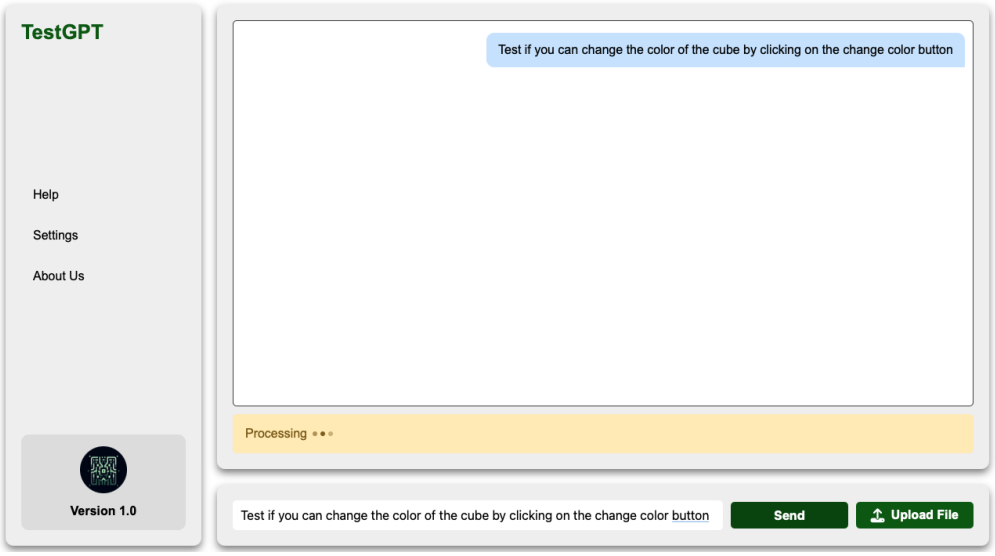

Figure 5.9: The user's test scenario and the Unity scene file will be sent to ChatGPT.

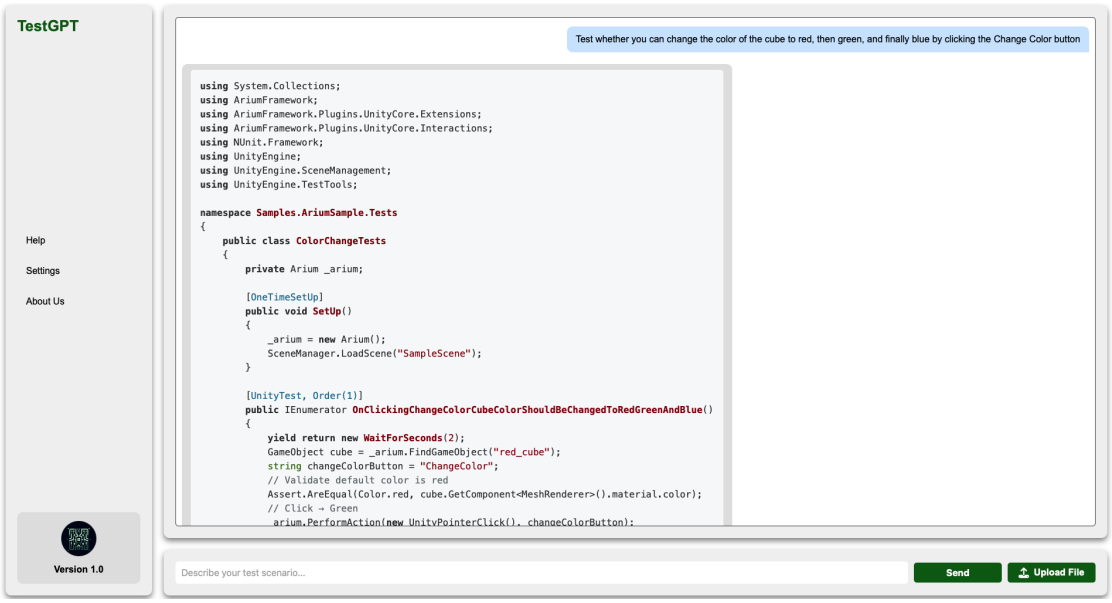

Figure 5.10: Users can see the full generated code by ChatGPT O3 mini and the history of the chat.

```
[UnityTest, Order(1)]
public IEnumerator OnClickingChangeColorCubeColorShouldChange()
{
    yield return new WaitForSeconds(2);
    GameObject cube = _arium.FindGameObject("red_cube");
    string changeColorButton = "ChangeColor";
    // Validate default color is red
    Assert.AreEqual(Color.red, cube.GetComponent<MeshRenderer>().material.color);
    // Click → Green
    _arium.PerformAction(new UnityPointerClick(), changeColorButton);
    yield return new WaitForSeconds(1);
    Assert.AreEqual(Color.green, cube.GetComponent<MeshRenderer>().material.color);
    // Click → Blue
    _arium.PerformAction(new UnityPointerClick(), changeColorButton);
    yield return new WaitForSeconds(1);
    Assert.AreEqual(Color.blue, cube.GetComponent<MeshRenderer>().material.color);
}
```

Figure 5.11: Example of the generated code based on the Arium framework for the test scenario: changing the color of the cube to green and blue by clicking on the Change color button.

5.3.1 Prompt Engineering

One of the key components in the development of TestGPT was prompt engineering. Given the user's test scenario and the Unity scene file, it is essential to construct a well-designed prompt to send to ChatGPT. Since TestGPT needs to generate output code in the Arium framework, it is necessary to provide the model with the syntax and structure of the framework. After examining the results of the generated code when only passing the syntax of Arium, we realized that including examples helped ChatGPT produce more accurate code. Therefore, in the final prompt, we provide three examples showing the structure, required packages, use of assertions, and other details in the output code. With this information, the o3-mini model is able to reason and analyze the provided examples, interpreting the user's scene file and test scenario, and generating context-aware test code accordingly.

To enhance the accuracy of LLM-generated test scripts in TestGPT, we implemented a dual-agent architecture inspired by the concept of having a controller agent. This design addresses one of the challenges in code generation using large language models (LLMs): although LLMs can produce syntactically valid code, the output often contains semantic or logical errors that can cause test failures or undesired behavior. Rather than relying solely on human intervention to identify and fix these errors, we introduced a second LLM—referred to as the controller agent—to act as an automated code reviewer.

In this architecture, the first agent is responsible for generating test code based on natural language test scenarios and scene information. Once the initial code is generated, it is passed to the second agent, whose task is to validate and refine this output. The prompt engineering for the second agent differs from that of the first. Instead of asking it to generate new code, we prompt it to perform structured checks on the generated code, including verifying imported libraries, identifying syntax and semantic issues, and reasoning about potential logical flaws.

In addition to these two agents, we employ a third agent, known as the checker agent, whose role is to determine whether a user's prompt is related to test generation. Since sending data to ChatGPT is costly, it is important to filter out unrelated prompts before passing them to the main system. The prompt of the checker agent is simple: verify whether the user's request is relevant to test generation or not.

The prompt sent to the first agent contains the user's test scenario, the Unity scene file (YAML file), the syntax of the Arium framework, and three examples. Each example includes a test scenario along with its corresponding test function that we wrote manually, which provides the model with a clear template for code generation.

The prompt sent to the second agent instructs it to check the following aspects of the generated code:

- verifying the libraries included in the code,

- checking the syntax of the generated code,
- ensuring that all interactions requested by the user are covered,
- confirming that the generated assertions align with the user's test scenario.

If the second agent identifies issues in the code, it attempts to fix them. If no changes are necessary, the generated test script is returned to the user, who can then copy it into the Unity test files and execute it.

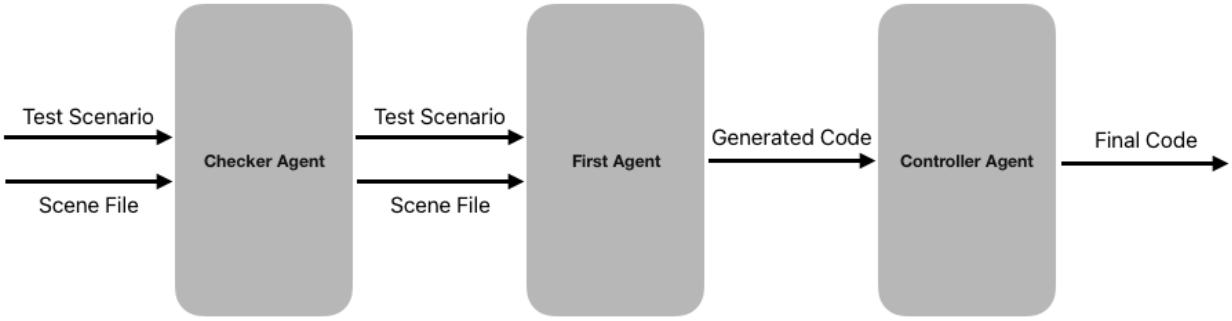

Figure 5.12: Workflow of these agents working together to generate test scripts based on the user's input and the Unity scene file.

```
ATRIUM_SYNTAX = r"""
Arium Framework Syntax Reference:
1. Instantiate Arium:
   var _arium = new Arium();
2. Find GameObjects:
   _arium.FindGameObject("Name");
3. Get Components:
   _arium.GetComponent<Component>("Name");
4. Perform Actions:
   _arium.PerformAction(new UnityPointerClick(), "Name");
5. Unity Event System:
   UnityEventSystemInteraction<T>.PerformAction("Name");
"""
```

Figure 5.13: Provide the simple syntax of the Arium in the prompt

5.3.2 Testing with TestGPT

TestGPT can analyze the Unity scene file uploaded by the user, which means it has access to the attributes of the objects within the scene. In each testing scenario, we typically check the values of these attributes either

```
[UnityTest, Order(3)]
public IEnumerator VerifyCubePositionIsZeroByDefault()
{
    yield return new WaitForSeconds(2);
    GameObject cube = _arium.FindGameObject("Cube");
    Assert.AreEqual(Vector3.zero, cube.GetComponent<Transform>().position);
    yield return null;
}
```

Figure 5.14: Provide the simple test function with Arium in the prompt

after an action that modifies them or at the initial stage of the software before any actions are performed. By having access to these attributes and the ability to use assertions, we can technically handle any test related to the properties of objects in the scene. However, there are other aspects of testing, such as performance testing (e.g., frame rate or resolution), that are not supported by this tool.

5.3.3 Excluding Memory in TestGPT

In designing *TestGPT*, the system was intentionally implemented without conversational memory to ensure that each generated test script was produced in isolation from prior interactions. This decision was made to improve the reproducibility and fairness of the evaluation. Because large language models may generate slightly different outputs across sessions due to their probabilistic nature, maintaining conversational memory could introduce subtle biases influenced by previous prompts, feedback, or context accumulation. By resetting the model's state for each generation, all participants interacted with a consistent system that responded solely to the current input scenario and Unity scene data.

5.4 GameDriver Testing Tool

5.4. GAMEDRIVER TESTING TOOL

GameDriver [66] is a cross-platform test automation framework designed to support a wide range of gaming and XR applications. It works seamlessly with game engines such as Unity and Unreal Engine, enabling developers and QA engineers to execute consistent, repeatable functional tests across platforms including PC, mobile, VR/AR, and consoles. At its core, GameDriver uses a patented agent-based architecture that allows users to simulate various inputs—ranging from keyboard and mouse to touch, gamepads, and XR controllers such as those used with the Meta Quest. Importantly, the framework does not require instrumentation or modification of the underlying application code. Testers can drop the agent into the game build and immediately begin authoring and executing test scenarios using GameDriver's intuitive API.

What makes GameDriver particularly powerful is its Hierarchy Path query language, a structure similar to XPath used in web testing, which enables precise querying and manipulation of in-game objects. Through this system, testers can perform a wide variety of actions—clicks, drags, touch events, axis movements, method invocations, and more. The API also provides extensive control over object behaviors, such as reading or setting values for position, rotation, and other properties at runtime. Developers can execute both public and private in-game methods without direct code access, enhancing flexibility and supporting black-box testing scenarios. GameDriver also supports utility functions like scene loading, FPS monitoring, object caching, and collision detection—making it especially suitable for large, complex scenes and performance-sensitive applications.

GameDriver captures the inputs and interactions that occur within a scene and allows testers to write tests based on those interactions. By recording these interactions, it becomes possible to repeat the same sequence of actions and include assertions to compare the final output with the expected result, thereby validating the functionalities involved in the interaction sequence.

For example, testers can simulate picking up a key in VR, placing it in a drawer, and verifying that the drawer opens as expected—without manually repeating the steps in the headset. The ability to automate XR interaction sequences such as teleportation, object manipulation, and environmental feedback makes GameDriver particularly valuable for streamlining repetitive tasks, as it eliminates the need for manual repetition during each test cycle.

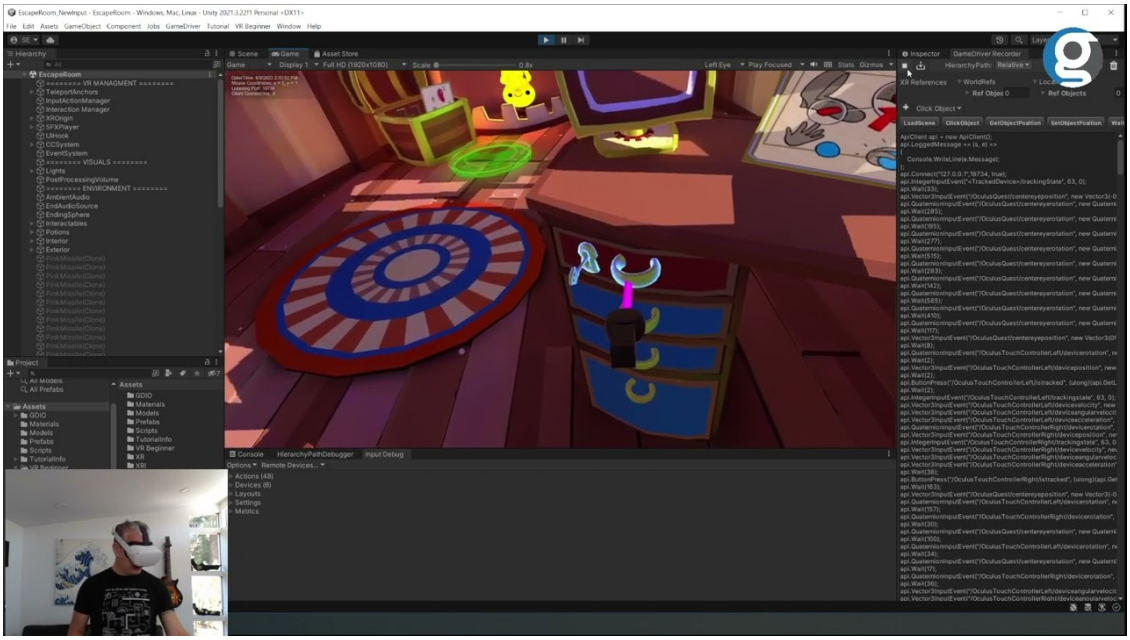

Figure 5.15: The GameDriver recorder records user interactions for replaying later [66].

```
[Test]
public void ReplayAllCubeActions()
{
    api.Click(MouseButtons.LEFT, 827f, 448f, 58, 30);
    api.Wait(986);
    api.Click(MouseButtons.LEFT, 827f, 448f, 74, 30);
    api.Wait(864);
    api.Click(MouseButtons.LEFT, 827f, 448f, 57, 30);
    api.Wait(842);
    api.Click(MouseButtons.LEFT, 827f, 448f, 43, 30);
    api.Wait(2719);
    api.Click(MouseButtons.LEFT, 834f, 444f, 64, 30);
    api.Wait(490);
    api.Click(MouseButtons.LEFT, 834f, 444f, 74, 30);
}
```

Figure 5.16: An example of interactions captured from the scene.

```
using System;
using NUnit.Framework;
using gdio.unity_api.v2;
using gdio.common.objects;

namespace TwoCubeTests
{
  [TestFixture]
  public class TwoCubeScenarios
  {
    private ApiClient api;

    [OneTimeSetUp]
    public void Connect()
    {
      api = new ApiClient();
      api.Connect("localhost", 19734, true, 30);
      api.EnableHooks(HookingObject.ALL);
    }

    [Test]
    public void ReplayAllCubeActions()
    {
      api.Click(MouseButtons.LEFT, 827f, 448f, 58, 30);
      api.Wait(986);
      api.Click(MouseButtons.LEFT, 827f, 448f, 74, 30);
      api.Wait(864);
      api.Click(MouseButtons.LEFT, 827f, 448f, 57, 30);
      api.Wait(842);
      api.Click(MouseButtons.LEFT, 827f, 448f, 43, 30);
      api.Wait(2719);
      api.Click(MouseButtons.LEFT, 834f, 444f, 64, 30);
      api.Wait(490);
      api.Click(MouseButtons.LEFT, 834f, 444f, 74, 30);

      // After performing the recorded actions, you can add assertions here
      // to check any parameters or attributes you want.
    }

    [OneTimeTearDown]
    public void Disconnect()
    {
      api.DisableHooks(HookingObject.ALL);
      api.Disconnect();
    }
  }
}
```

Figure 5.17: Testers can use recorded interactions and then add assertions at the end to verify the conditions they want in the test scenario

Chapter 6

Data Analysis, Discussion, and Limitation

This chapter includes an overview of the primary parameters investigated in the study, followed by a comprehensive analysis of the collected data, a detailed discussion of the findings, and an examination of the study's limitations. The analysis integrates both quantitative and qualitative data gathered from participants.

6.1 Explanation of Metrics and Calculations

In this thesis, we used various metrics and parameters to analyze the collected data and find meaningful insights. To evaluate and compare the effectiveness of different methodologies, we considered following metrics:

- Task Completion: Number of participants who finished all testing tasks for each phase.
- Test Quality: Number of bugs found in each task.
- Test Development Performance: Calculated as the number of tests completed per minute, indicating efficiency.
- System Usability Scale (SUS): A validated, standardized questionnaire assessing perceived usability on a scale from 0 (poor) to 100 (excellent), with scores above 68 indicating acceptable usability.
- ANOVA (Analysis of Variance): A statistical test used to determine if observed differences among group means are statistically significant.

- Pearson Correlation Coefficient (r): Measures the strength and direction of linear relationships between variables, with values ranging from -1 (strong negative) to +1 (strong positive).

6.2 Task Completion

In each testing phase, participants were asked to complete six different test scenarios. Each phase was limited to 20 minutes, and one of the key parameters we examined was the number of participants who were able to complete all tasks within this time. This parameter is important, as one of the main goals of these methodologies is to save time during the testing process.

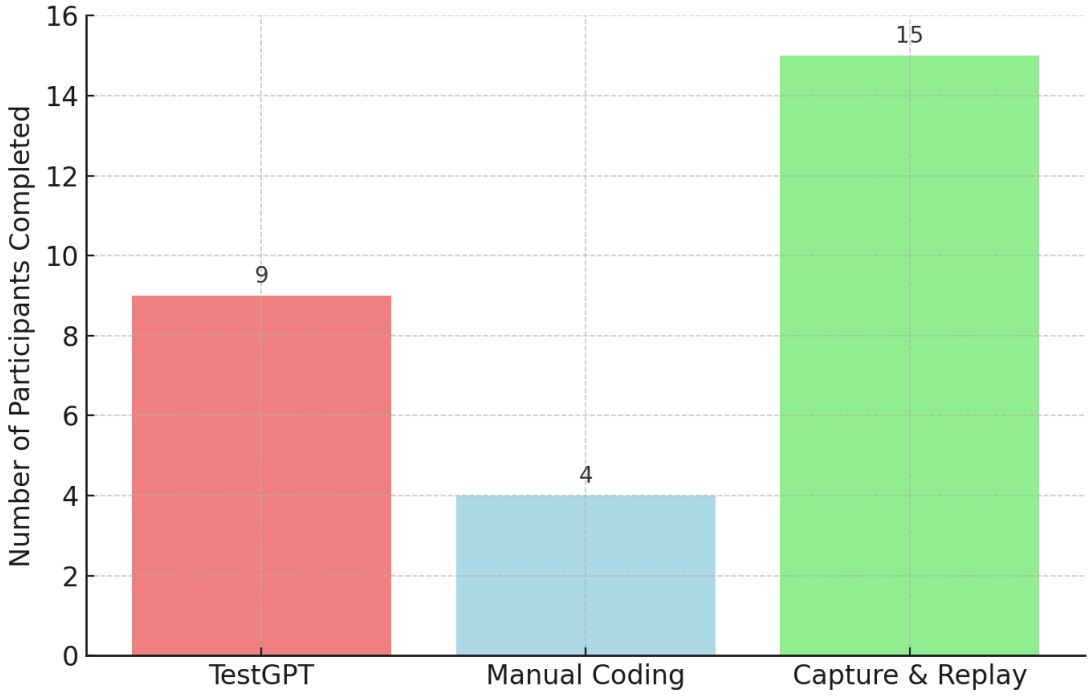

Figure 6.1: Task Completion Comparison Between Testing Methodologies

All participants were able to complete all tasks using the capture-and-replay methodology. In contrast, 60% of participants completed all tasks using TestGPT by describing test scenarios in natural language, while 26% of participants managed to complete all tasks using manual coding. These results indicate that writing test scripts manually is more time-consuming compared to describing scenarios in natural language or recording and replaying actions. Capture-and-replay, on the other hand, achieved the best results, with every participant completing all tasks.

6.3 Test Quality

In each testing scenario, we intentionally put bugs, and one of the parameters we examined was the number of bugs detected by participants, as one of the main goals of testing is to identify defects in order to resolve them. For this parameter, all participants successfully identified every bug inserted into the tasks. So, this metric does not provide a basis for comparing the effectiveness of the different testing methodologies.

6.4 Test Development Performance

Performance metrics illustrate the efficiency of each testing methodology. Table 6.1 summarizes the descriptive statistics for the three testing approaches evaluated in this study.

Method	**Mean Performance**	**Std. Deviation**	**Minimum**	**Maximum**
TestGPT	0.307	0.122	0.150	0.600
Manual Coding	0.204	0.117	0.050	0.400
Capture and Replay	0.797	0.166	0.500	1.000

Table 6.1: Test Development Performance Across Testing Methodologies

As shown in Table 6.1 and Figure B, the Capture-and-Replay testing method yielded the highest mean performance score, followed by TestGPT and Manual Coding.

Because all participants performed the same testing tasks using each of the three methodologies (within-subjects design), a one-way repeated-measures ANOVA was used to evaluate differences in test development performance. This statistical approach accounts for the correlation between repeated measures from the same participants and isolates the effect of the testing methodology itself, providing a more accurate assessment than a traditional between-subjects ANOVA.

Before running the analysis, normality was assessed using the Shapiro–Wilk test for each condition, which confirmed that performance scores were approximately normally distributed ($(p > 0.05)$ for all methods). Mauchly's test of sphericity indicated that the assumption of sphericity was not violated ($p > 0.05$), allowing standard F-ratios to be interpreted directly.

The repeated-measures ANOVA revealed a statistically significant main effect of testing methodology on performance, $F(2, 28) = 80.09$, $p < .0001$, $\eta^2 = 0.85$, indicating a large effect size. Post-hoc pairwise comparisons with Bonferroni correction showed that Capture-and-Replay significantly outperformed both TestGPT ($p < .001$) and Manual Coding ($p < .001$), while TestGPT also performed significantly better than Manual Coding ($p < .05$).

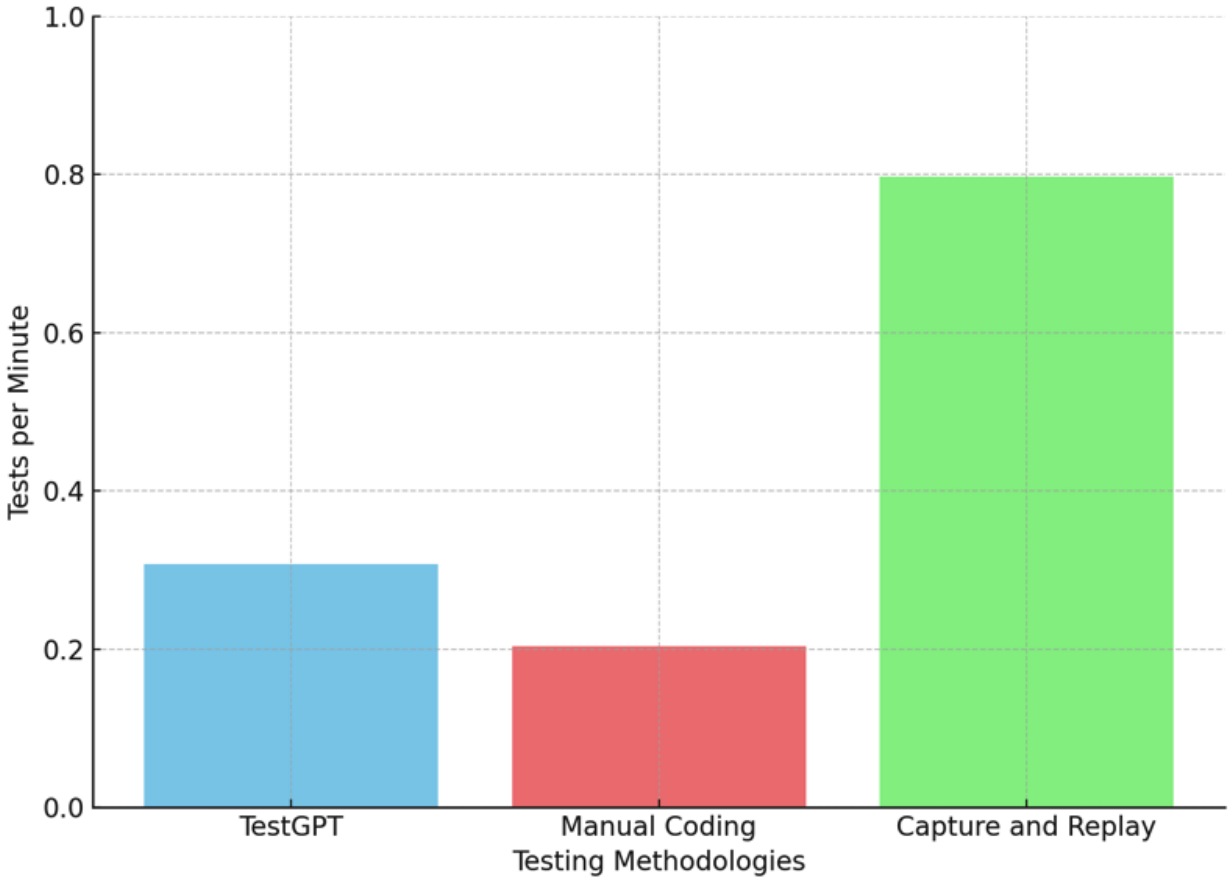

Figure 6.2: Performance Comparison Between Testing Methodologies

These findings demonstrate that the observed performance differences are not due to random variation but reflect genuine efficiency advantages of certain testing approaches. Specifically, the Capture-and-Replay method produced the highest performance levels overall, highlighting its strength in speed and repeatability. TestGPT offered moderate efficiency gains over manual coding, particularly for simpler test cases, confirming its potential as a practical tool for rapid test script generation in XR development workflows.

6.5 Usability Evaluation (SUS Scores)

SUS scores measure the user-perceived ease of use and satisfaction [67]. the followng table summarizes these scores. The System Usability Scale (SUS) is computed based on responses to 10 questions rated on a 5-point Likert scale. The questions alternate between positive (odd-numbered) and negative (even-numbered) statements. The SUS score is calculated using the following steps:

- For each odd-numbered question Q_i (i.e., questions 1, 3, 5, 7, 9), compute:

$$A_i = Q_i - 1$$

- For each even-numbered question Q_i (i.e., questions 2, 4, 6, 8, 10), compute:

$$A_i = 5 - Q_i$$

- Sum all adjusted scores:

$$\sum_{i=1}^{10} A_i$$

- Multiply the result by 2.5 to scale it to a 0–100 range:

$$\text{SUS Score} = \left(\sum_{i=1}^{10} A_i \right) \times 2.5$$

The final SUS score ranges from 0 to 100, with higher values indicating better perceived usability.

Method	Mean SUS Score	Std. Deviation
Arium	63.00	18.44
TestGPT	76.67	15.29
Capture and Replay	82.83	12.41

Table 6.2: SUS Scores by Testing Method

A one-way Repeated-Measures ANOVA was conducted to compare System Usability Scale (SUS) scores across the three testing methods. The results showed a statistically significant effect of testing method on SUS scores, $F(2, 42) = 5.94$, $p = 0.0053$, indicating that at least one group mean differed from the others.

To identify where the differences lay, Tukey's HSD post-hoc[87] test was performed. The analysis revealed that both TestGPT and Capture and Replay achieved significantly higher SUS scores than Arium. However,

the difference between TestGPT and Capture and Replay was not statistically significant, suggesting that participants perceived both as similarly usable, while Arium was rated lower.

Comparison	Mean Difference	p-value	Significance
TestGPT vs. Arium	13.67	0.041	*
Capture & Replay vs. Arium	19.83	0.004	**
Capture & Replay vs. TestGPT	6.16	0.524	n.s.

Table 6.3: Post-hoc Tukey HSD results for SUS score comparisons across testing methods

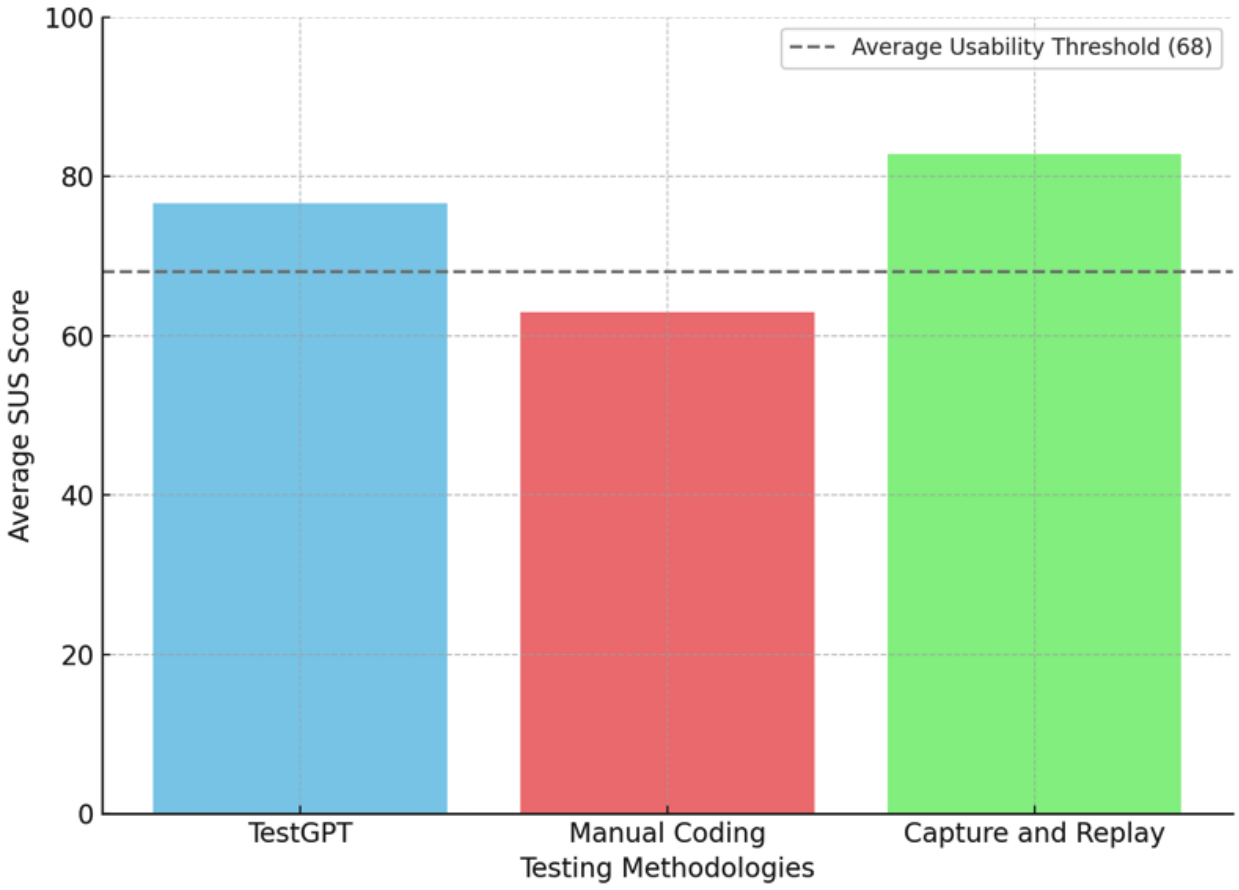

Figure 6.3: Usability Comparison Between Testing Methodologies.

Correlation analyses evaluated if higher usability perceptions correlated with better performance. No significant correlations emerged, suggesting that high usability scores do not inherently predict increased testing efficiency.

6.6 Participant Skills and Experience Impact Analysis

We anticipated that individuals with higher levels of XR and software testing experience would perform better. Following table explores correlations between participant experience and performance.

To evaluate the strength of linear relationships between participant skill levels and performance, the Pearson correlation coefficient (r) was computed to calculate the Pearson Correlation Coefficient.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where x_i and y_i represent individual values for the two variables, and $\bar{x}$ and $\bar{y}$ are their respective means.

To assess whether the observed correlations were statistically significant, a t-statistic was calculated to determine the p-value.

$$t = r \cdot \sqrt{\frac{n-2}{1-r^2}}$$

and the corresponding p-value was obtained from the t-distribution with $n-2$ degrees of freedom.

Relationship	**Pearson r**	**p-value**
XR Skill vs TestGPT Performance	-0.104	0.711
XR Skill vs Manual Performance	0.230	0.409
XR Skill vs CaptureReplay Performance	-0.314	0.255
Testing Skill vs TestGPT Performance	0.149	0.597
Testing Skill vs Manual Performance	0.272	0.327
Testing Skill vs CaptureReplay Performance	-0.154	0.584
Arium Experience vs Manual Performance	-0.188	0.503
GenAI Experience vs TestGPT Performance	0.243	0.383
CaptureReplay Experience vs Performance	0.093	0.740

Table 6.4: Correlation between Participant Skills/Experience and Performance

The correlation analysis did not reveal any statistically significant relationships between participants' background experience and their performance across the three testing methodologies. Although small positive trends were observed between XR skill and manual performance, testing skill and both manual and TestGPT performance, as well as generative AI experience and TestGPT performance, none of these reached statistical significance ($pvalues > 0.05$). Similarly, weak negative associations appeared between XR skill and CaptureReplay performance and between Arium experience and manual performance, but these too

were not significant. Overall, these results suggest that prior experience with XR development, software testing, or specific tools such as Arium, generative AI, or CaptureReplay did not have a measurable effect on performance in this study.

6.7 NASA-TLX Analysis

In addition to performance and usability, we examined perceived mental and physical demands, performance satisfaction, and frustration using NASA-TLX metrics. These values are rated from 1 (lowest) to 10 (highest).

Method	**Mental Demand**	**Physical Demand**	**Performance**	**Frustration**
Arium	5.60	2.60	6.53	3.33
TestGPT	3.20	1.20	6.80	2.80
Capture and Replay	2.53	1.53	8.80	1.53

Table 6.5: NASA-TLX Average Scores by Method

These results provide critical insights:

- Mental Demand: Capture and Replay had the lowest cognitive load (2.53), indicating that it required the least mental effort to complete tasks. In contrast, Arium imposed the highest mental demand (5.60), potentially due to manual complexity.
- Physical Demand: All methods scored relatively low, with Capture and Replay (1.53) and TestGPT (1.20) showing minimal physical effort.
- Performance Satisfaction: Participants felt most successful with Capture and Replay (8.80), suggesting confidence and efficiency. Arium received the lowest score (6.53).
- Frustration: Capture and Replay caused the least frustration (1.53), while Arium generated the most (3.33), further indicating that manual testing methods were more mentally and emotionally taxing.

These findings highlight that in addition to being the most efficient and usable method, Capture and Replay also provided the most comfortable and satisfying user experience across all dimensions.

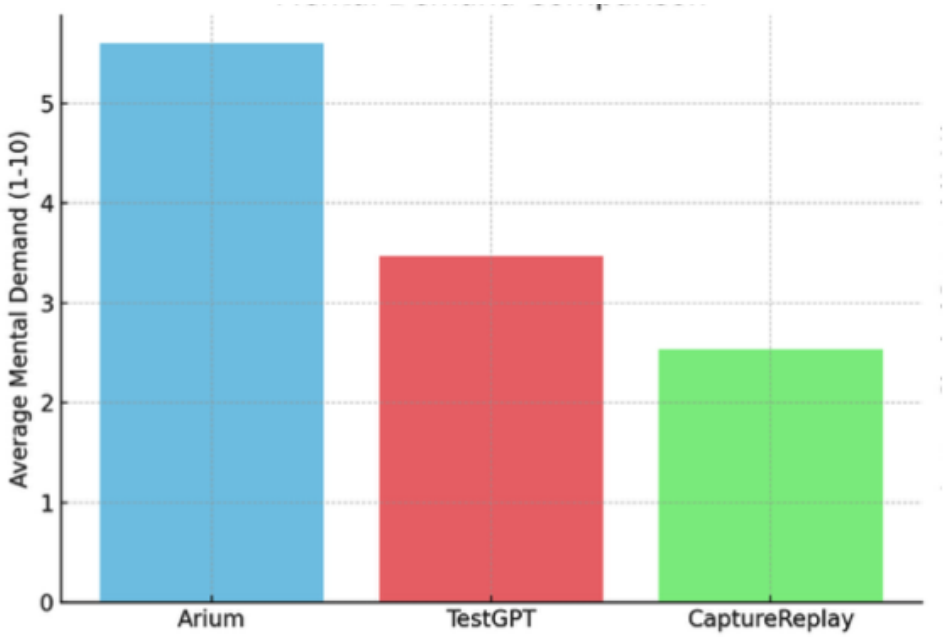

Figure 6.4: Mental Demand Comparison Between Testing Methodologies.

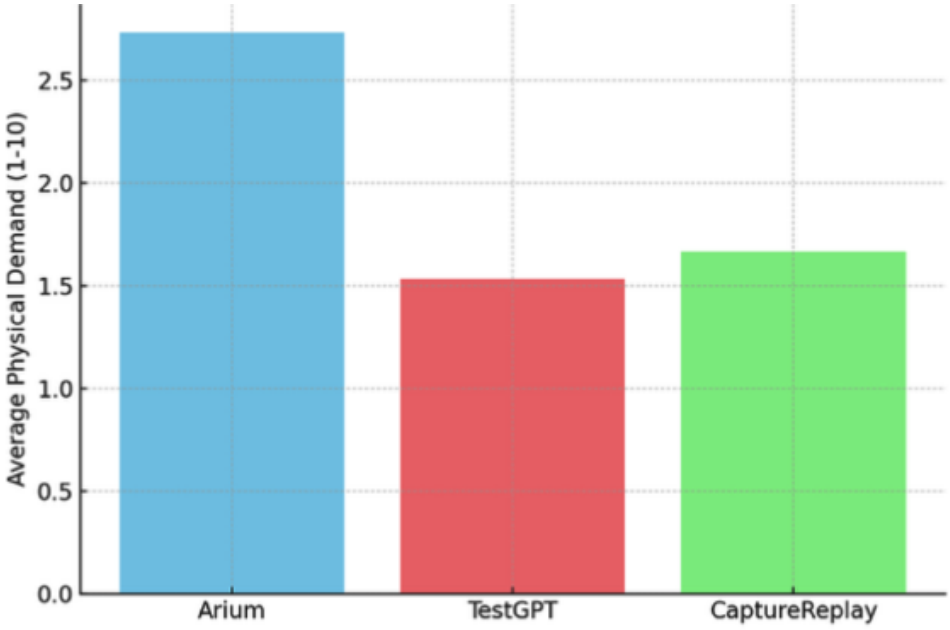

Figure 6.5: Physical Demand Comparison Between Testing Methodologies.

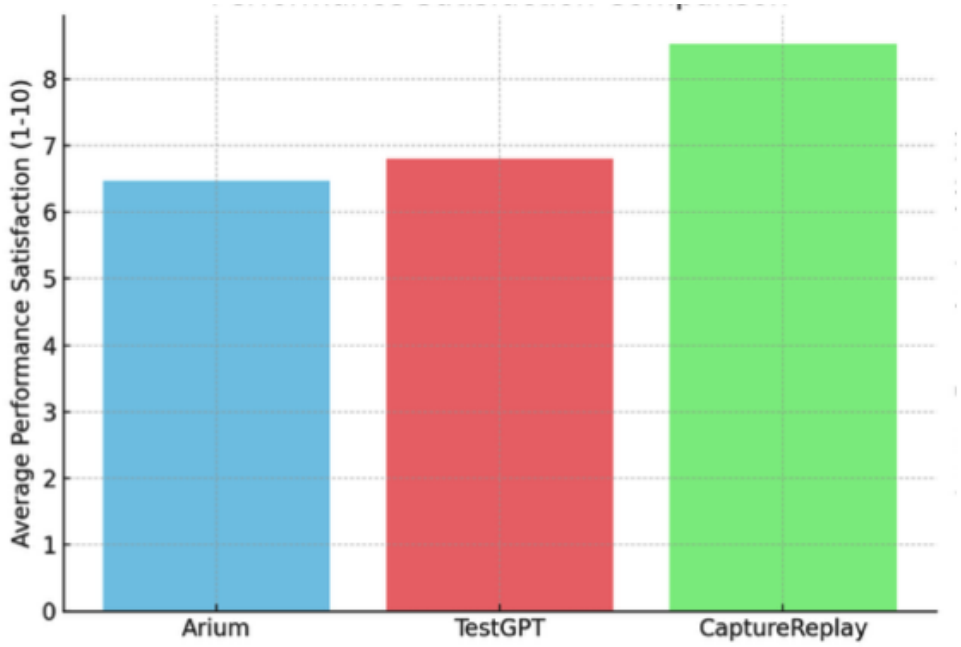

Figure 6.6: Performance Satisfaction Comparison Between Testing Methodologies.

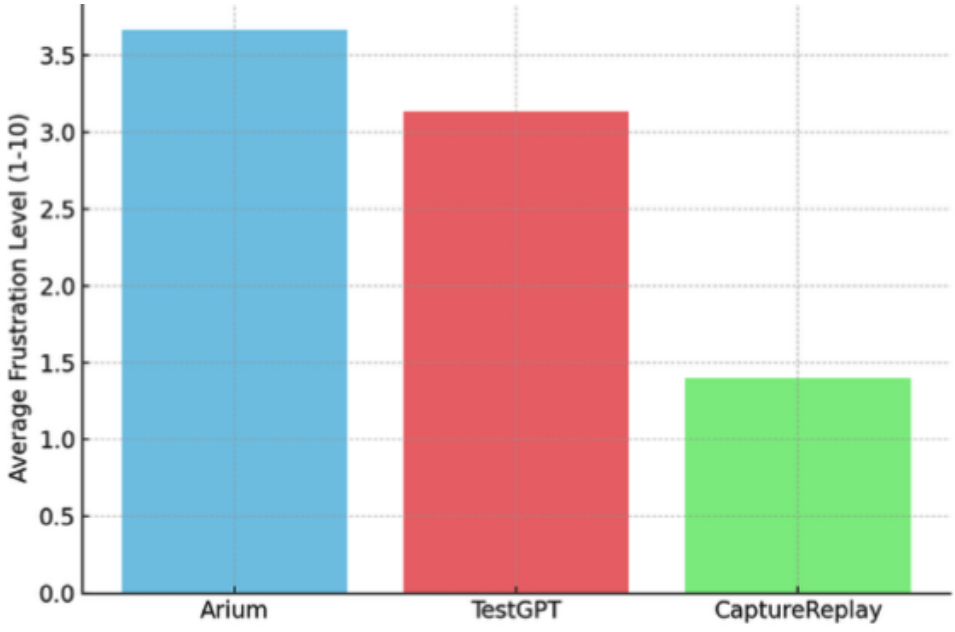

Figure 6.7: Frustration Level Comparison Between Testing Methodologies.

6.8 Analyzing Qualitative Data

After conducting the study and collecting the quantitative data, we also interviewed participants to discuss their experiences. We asked about the problems they encountered, ideas they had, anything they found interesting, or any issues they wanted to highlight regarding any of the testing methods they used. The purpose of these interviews was to gain deeper insights into their experiences. In this section, we present some of the key points mentioned during the interviews.

One of the things we asked participants was to rank the three different testing methodologies based on their personal preferences. It is important to note that liking a testing methodology is different from it being the most optimized or usable. This part of the study focused more on the participants' subjective interest after using all the testing methods. We asked them to provide their preferences for both simple and complex projects, and the results are presented below.

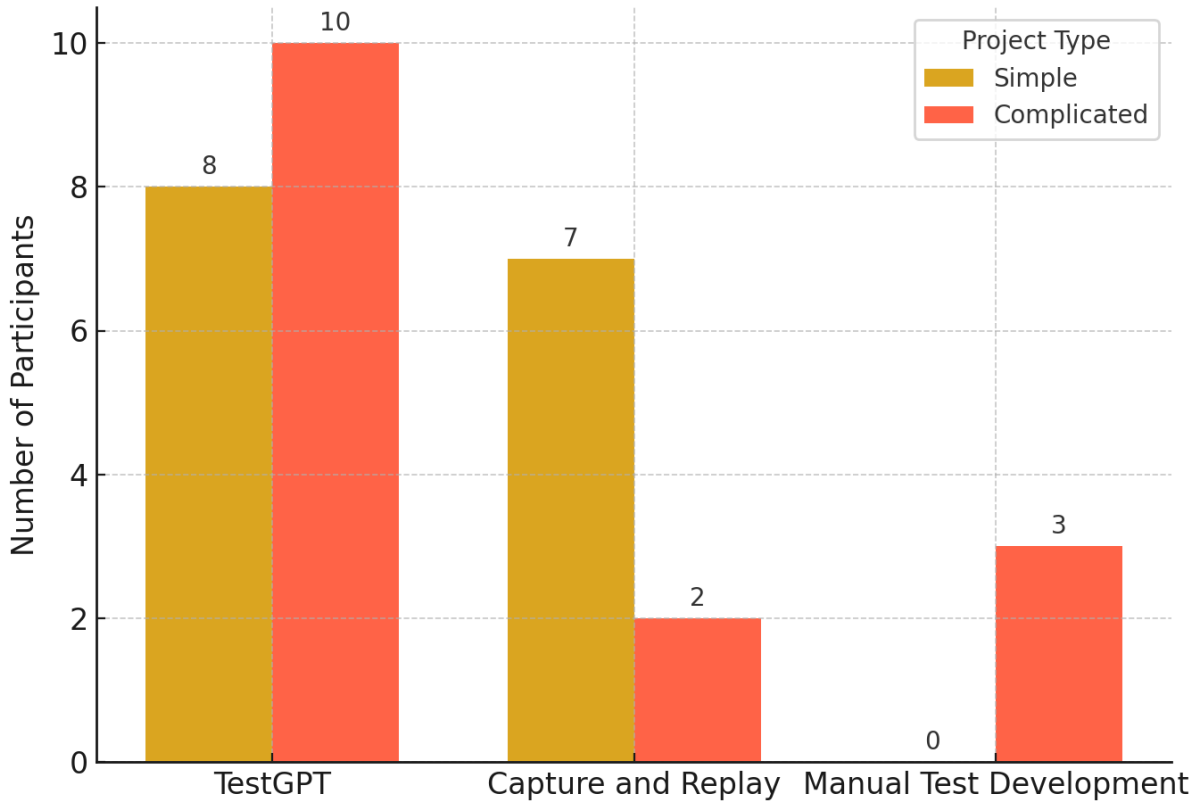

Figure 6.8: Preferred Testing Method by Project Type

Testing Method	Simple Project	Complicated Project
Manual Coding	0	3
TestGPT	8	10
Capture and Replay	7	2

Table 6.6: Participant Preferences for Testing Methods by Project Type

6.9 Qualitative Insights

Interviews with participants revealed several reflections on the practical use, strengths, and limitations of the three testing methodologies: manual coding using Arium, generative AI–based test generation using *TestGPT*, and capture-and-replay using GameDriver. These qualitative perspectives complement the quantitative findings by providing a human-centered view of how participants experienced each tool, their expectations, and the perceived realism of each approach in professional testing workflows.

6.9.1 Manual Coding with Arium

One notable theme was the sense of enjoyment and control participants felt when coding manually. They described the process as "fun" and said it gave them "a feeling of being in command" over the testing process. This level of control was particularly valued when handling precise logic or when verifying that the test covered specific functional paths. As one participant remarked, "When I write the code myself, I know exactly what it's doing, line by line."

However, several participants also noted that this precision came at a cost of scalability and efficiency. Maintaining test scripts for larger projects is challenging: "Writing code for a big project would be a big project by itself." Others pointed out that while manual coding felt rewarding, it would not be feasible under real-world production deadlines. As one participant summarized, "If we had 50 buttons instead of six, this would take forever." Despite these limitations, manual testing was consistently seen as the "most reliable" method, since testers could "trust what they had written."

6.9.2 Generative Testing with TestGPT

Generative testing via *TestGPT* was widely appreciated for its simplicity and natural-language interface, especially during the early stages of testing. Participants repeatedly emphasized the accessibility of describing tests in plain English rather than coding them. "It's much easier to just say what I want the test to do

instead of remembering the Arium syntax," one participant noted. Another explained, "It felt like talking to someone who understands what I mean until the scenario got too complicated."

However, complexity was a clear boundary for most users. When describing conditional logic, object dependencies, or spatial relationships, participants found it difficult to provide complete instructions in a single prompt. "It's easy when the task is small," said one participant, "but when I tried to explain the full workflow, it didn't understand some parts." The lack of conversational memory further amplified this issue, as users had to retype or slightly modify long prompts. "Having chat history would help a lot," one participant reflected. "It was frustrating to rewrite the same description just to add one more condition."

Some users also commented on errors in the generated code, although these were primarily related to mismatched object names rather than logical or syntactical mistakes. "It's not that the logic is wrong—it just doesn't find the right cube," one tester explained. This observation reinforced the importance of clear scene descriptions and consistent object naming, as generative models rely heavily on textual context for accurate code mapping. A few participants even experimented by rephrasing their prompts, noting that "slight wording changes sometimes fixed the issue," highlighting both the adaptability and unpredictability of LLM-based generation.

6.9.3 Capture-and-Replay with GameDriver

The capture-and-replay method was praised for its intuitive and visual workflow. Participants appreciated being able to "just record" interactions instead of scripting them. "It's like using a screen recorder for testing, it feels natural," one participant said. This made it particularly effective for user interface–driven scenarios such as games or simulation prototypes. In collaborative settings, some participants envisioned practical benefits: "It's good to have some tests recorded and then run them to see if everything still works after a merge." Another added, "You can even show a bug to your teammate without explaining it, just play it back."

However, users also identified important drawbacks. Many noted that the approach was "too sensitive to layout changes," meaning even small interface adjustments could invalidate recorded tests. "If I move a button two pixels, the test breaks," one participant explained. Others recommended that the tool should use object identifiers instead of coordinates: "It would be better if it tracked the object by name, not by position."

6.9.4 Comparative Reflections and Hybrid Preferences

Across the three methods, a recurring theme was the trade-off between control and convenience. Manual coding offered precision but required effort; capture-and-replay was quick but fragile; and *TestGPT* provided automation and flexibility but struggled with complex prompts. Several participants envisioned a hybrid solution that could merge the best of these approaches: "If we could record something and then have the AI write or fix the code, that would be perfect." Another suggested, "Maybe use AI for generating the first version, and then capture tools to verify it visually."

In summary, the qualitative feedback revealed that:

- **Manual coding** is favored for control and precision but becomes less scalable as project complexity increases.
- **TestGPT** is efficient and user-friendly for simple scenarios but challenged by complex contexts and lack of conversational memory.
- **Capture-and-replay** is intuitive, visual, and well-suited for regression testing but sensitive to UI changes and difficult to debug.

6.10 Limitations

6.10.1 Complexity of the Projects

One of the key limitations of this study was the level of complexity in the test scenarios. As outlined in the data analysis and reflected in participant feedback, testing a simple project—with basic interactions such as clicking on a few static objects is considerably different from testing complex, dynamic XR environments. Real-world XR projects often involve asynchronous behaviors, event-driven triggers, continuous tracking, and high variability in object states. In this study, participants worked within a constrained and intentionally simplified Unity scene featuring two cubes and six interaction buttons. Although this setup effectively isolated functional behaviors and facilitated controlled comparisons, it did not fully represent the scope and intricacy of production-level XR applications. Consequently, the generalizability of these findings to larger or more realistic XR systems remains limited. Future evaluations should examine how the tested methodologies perform when applied to richer 3D environments with multiple interactive components, concurrent user actions, and greater data dependencies.

6.10.2 Learning Curve for New Frameworks

Another limitation relates to the participants' unfamiliarity with the testing methodologies. Regardless of whether they used Arium, TestGPT, or GameDriver, each tool represented a new workflow for most participants. Using a method immediately after instruction introduces a learning curve that can affect both efficiency and accuracy. Participants may have overlooked tool features, struggled with syntax, or required additional time to interpret feedback. Although none of these tools are inherently complex, the limited practice time likely affected performance outcomes. Longer term studies, where participants can gain proficiency over days or weeks, would provide a fairer assessment of each tool's learning demands and potential productivity in sustained use.

6.10.3 Learning Effects and Task Order Bias

A related design constraint involved the fixed order in which participants completed the testing tasks and addressed software bugs. All participants were instructed to perform the six test scenarios in the same predefined sequence, beginning with color and position changes and ending with scaling operations. This consistent order ensured comparability but may have introduced learning effects. For instance, participants might have become more efficient or confident after completing earlier tasks, leading to faster performance on later ones independent of the methodology used.

6.10.4 Merging Test Scenarios

In practice, testers often combine multiple test scenarios into a single case to streamline workflows. However, in this study, participants were instructed to execute scenarios individually and in sequence. While some participants experimented with merging cases after completing the individual ones, this introduced variability in performance measurements. For example, merging "increase depth" and "decrease depth" into one script affected how time and complexity were measured, making comparisons inconsistent. Such variation reduces the reliability of quantitative conclusions and highlights the need for more structured test execution protocols in future work.

6.10.5 Limited Sample Size and Generalizability

The study included 15 participants, which, while sufficient for exploratory analysis, limits the statistical power and generalizability of the results. Individual differences in technical experience, familiarity with Unity, or comfort with automation tools may have disproportionately influenced the findings. Additionally, the participant pool consisted mainly of students and researchers rather than professional QA engineers or

industry practitioners, which may affect external validity. The simplicity of the tested application, combined with this limited and relatively homogeneous participant group, means that conclusions should be interpreted as preliminary indicators rather than definitive measures of tool performance. Expanding future studies to include a larger and more diverse population—spanning professional testers, developers, and XR designers—would strengthen the robustness and applicability of the results.

6.10.6 Limitations in Testing Methodologies

Each testing methodology included in the study presented inherent constraints. In the case of TestGPT, the absence of chat history was a deliberate design decision to reduce participant bias and ensure reproducibility. However, this also made prompting less efficient, as participants had to rewrite or modify full prompts instead of iterating interactively. Several interviewees noted that a persistent memory would have streamlined testing and improved usability. Additionally, TestGPT was developed as part of this research and is still in an early stage compared to mature commercial tools like GameDriver. Therefore, differences in tool stability, polish, and feature completeness likely contributed to some usability gaps observed in the study.

6.10.7 Uncertainty in Using New Tools

Finally, participants encountered general uncertainty when using unfamiliar frameworks. Arium, TestGPT, and GameDriver each required new ways of thinking about automation, and participants often had limited prior exposure to these methods. Early-stage unfamiliarity can influence both confidence and performance, potentially underrepresenting each tool's full capabilities. Future work should include training or pre-study practice sessions to mitigate these early learning effects and to distinguish between tool usability and user adaptation challenges.

6.11 Discussion of Results

The results of this study show some clear differences between the three testing methods. Capture and Replay turned out to be the fastest and most consistent, with all participants finishing their tasks within the time limit and showing the highest performance scores. TestGPT came next, with around 60% of participants completing all tasks, and it was noticeably easier than Manual Test Development, where only about a quarter of participants finished. On usability, both TestGPT and Capture and Replay scored higher than Manual coding, and the difference between them was not significant. Bug detection was the same across all methods since everyone managed to find the bugs we added, and experience level did not seem to have a clear effect on performance.

The interviews helped explain these patterns. Participants liked the ease and speed of Capture and Replay, especially for simple user interactions and quick regression checks, although they pointed out that the method could break when the interface changed. TestGPT was appreciated for making it easy to create simple tests just by describing them in plain language, but for more complex scenarios, writing longer prompts without history was tiring. Manual coding was valued for the control it gave testers, but it was also seen as slower and harder to scale up. When asked about preferences, TestGPT and Capture and Replay were chosen more often for simple projects, while TestGPT was favored most for complex projects.

Looking across all findings, it seems that each method has its own strengths. Capture and Replay is best when tasks need to be done quickly and repeated often. TestGPT is helpful for creating tests in a natural and less technical way, especially for projects that get complicated. Manual coding is still important when very precise logic is needed, even though it takes more effort.

Creating test scripts using generative AI showed both benefits and limitations in our study. Quantitatively, it enabled higher task completion than manual coding and better development performance, while still remaining below capture-and-replay. Participants rated TestGPT highly on usability and reported low mental and physical demand with low frustration, indicating that describing tests in natural language reduces effort and speeds up simple test creation. Qualitatively, users valued its simplicity in the early stages but noted that complex scenarios required more precise, longer prompts; the absence of chat history increased re-prompting effort; and generated scripts sometimes failed due to mismatched object names rather than logic or syntax, making debugging feel harder than writing from scratch. Overall, TestGPT is a usable option for simple and moderately complex tests.

Chapter 7

Summary and Future Work

7.1 Summary

This thesis investigates and compares different automated testing methodologies for Unity-based XR applications, focusing on usability and performance. With the growing use of XR technologies across industries, ensuring the quality and reliability of these applications has become important. The study evaluates three testing approaches: Manual Test Development using the Arium framework, LLM-based test generation using TestGPT, and Capture and Replay technique using GameDriver.

The research begins with a comprehensive review of software testing methodologies, identifying the specific challenges associated with testing XR applications, such as spatial interactions, timing sensitivity, and sensor-based input. The study aims to assess whether emerging AI-assisted tools like TestGPT can streamline the test development process. Participants were introduced to all three methods and tasked with creating test cases for testing scenarios.

The experiment involved 15 participants with varying levels of XR development and software testing experience. Each participant used all three methodologies to complete predefined testing tasks. Their performance was evaluated based on the time taken, number of test cases completed, and system usability (SUS) and NASA-TLX scores. Additionally, participants ranked the tools for both simple and complex projects and provided qualitative feedback in follow-up interviews.

The results indicate that Capture and Replay had the highest usability and the lowest cognitive and physical demand, particularly for simple projects, making it the most user-friendly and efficient option. TestGPT, despite being innovative and preferred for complex projects due to ease of expressing scenarios in natural language, was limited by the absence of chat history and occasional code generation errors, which

made debugging more time-consuming. Manual Coding with Arium provided a high level of control but was more demanding, especially for participants unfamiliar with the framework.

Statistical analyses, including means, standard deviations, Pearson correlations, and p-values, were used to evaluate relationships between user experience, performance, and tool preference but no statistically significant correlations were found.

Interviews revealed further insights: participants found coding enjoyable and appreciated the control it provided, but acknowledged that it becomes time-consuming in large projects. The absence of history of chat in TestGPT led to redundancy in prompts, and participants emphasized the value of maintaining prompt history. Capture and Replay was especially appreciated for regression testing and showcasing bugs to other team members.

In conclusion, this thesis contributes to the field of software testing in XR by presenting a comparison of three testing methodologies through both quantitative and qualitative lenses. Each method demonstrated strengths and limitations depending on the task complexity and user experience level. While Capture and Replay excels in ease of use and reliability for straightforward scenarios, TestGPT shows promise for future integration if conversational memory and debugging capabilities improve. Manual coding remains essential for full control in sophisticated projects but may not scale efficiently without supportive frameworks.

7.2 Future Work

The gaps identified in the limitations section of this study provide valuable opportunities for future research. One key direction would be to extend this investigation into testing more complex XR applications. While the current study focused on relatively simple scenarios with limited interactions and object variability, future research should explore how these testing methodologies perform under more challenging conditions. For instance, test cases involving asynchronous behaviours, object state dependencies, time-based interactions, or significantly larger scenes could reveal the strengths and weaknesses of each approach. Comparing results across simple and complex environments may provide a more comprehensive understanding of each tool.

Another promising area involves conducting a long-term study where participants use the testing methodologies in the context of real-world development projects. Unlike a short lab-based evaluation, a longitudinal deployment over several weeks or months would allow participants to become more comfortable with each tool and integrate them naturally into their development workflows. This approach would help identify usability issues that only emerge with sustained use, provide insights into learning curves, and uncover how each tool supports continuous integration and regression testing.

Several opportunities also exist to improve the design and capabilities of *TestGPT*. For example, one

limitation in the current implementation was the absence of chat history, which forced users to rewrite full prompts instead of building iteratively. Future versions could incorporate persistent memory, allowing users to refine tests incrementally, correct code errors, and receive context-aware responses. This enhancement would improve productivity and align the tool more closely with real-world developer workflows. Additionally, refining prompt engineering strategies—such as using structured templates, example-driven instructions, or adaptive guidance—could reduce ambiguity and increase code consistency across generations. Future research may also integrate automated bug reporting pipelines, where failed test executions trigger TestGPT to analyze logs and propose potential fixes, effectively closing the loop between test generation and defect diagnosis.

Another direction involves extending *TestGPT* to support multiple large language models (LLMs). Although this study used a single model configuration for experimental consistency, the system architecture could easily be adapted to interface with alternative models (e.g., GPT-4o, Claude, Gemini, or open-source LLMs). Comparative studies could examine how differences in model architecture, token limits, and reasoning capabilities influence code accuracy, test coverage, and natural-language understanding. Making model selection configurable within the codebase would further promote reproducibility and transparency in future experiments, enabling researchers and practitioners to replicate or extend the study under controlled conditions.

Another promising line of development lies in combining capture-and-replay techniques with LLM-based code generation. While capture and replay excel in usability and regression support, they are limited in handling dynamic logic or conditional reasoning. Conversely, LLMs like *TestGPT* can generate adaptive test scripts that account for variable conditions and logical dependencies. A hybrid tool could allow users to record interaction flows visually and then annotate them using natural language to generate assertions or extend test logic. This integration would offer the precision of scripted testing with the accessibility of recording-based workflows, reducing manual overhead and enhancing scalability.

From an industry perspective, the findings of this research could inform the design of practical QA tools for XR development teams in sectors such as gaming, simulation, and enterprise training. QA engineers could leverage *TestGPT* to automatically generate regression tests for interactive scenes, validate object behaviors, or script automated scenario walkthroughs without extensive coding expertise. Startups and studios could integrate such tools into their continuous integration pipelines to accelerate testing cycles and reduce costs while maintaining high-quality immersive experiences.

Finally, from an academic standpoint, this research opens opportunities for interdisciplinary collaboration between AI and software engineering communities. Future doctoral-level studies could investigate adaptive learning systems that fine-tune LLMs using domain-specific XR data, or explore human–AI co-testing

frameworks that blend automated reasoning with human validation. By bridging the fields of generative AI, software testing, and XR development, this line of research contributes to both the practical and theoretical advancement of automated testing methodologies.

Future researchers can build upon this work by extending the open-source implementation of *TestGPT*, which is publicly available on GitHub.[1] The modular architecture of the system allows easy substitution or integration of different large language models, enabling comparative studies across emerging models with varying reasoning and code-generation capabilities. Researchers can adapt the model configuration section of the code to switch providers or fine-tuned variants, modify the prompt construction logic to improve clarity or domain specificity, or implement new features such as persistent memory, context caching, or adaptive feedback mechanisms. This openness encourages collaborative experimentation, reproducibility, and iterative improvement, allowing the system to evolve alongside advances in generative AI and XR testing technologies.

[1] Available at: https://github.com/NamiMod/testgpt_deploy

Bibliography

[1] Introducing Apple Vision Pro: Apple's first spatial computer https://www.apple.com/ca/newsroom/2023/06/introducing-apple-vision-pro/. Accessed: 2024-07-17.

[2] Android XR: The Gemini era comes to headsets and glasses. https://blog.google/products/android/android-xr/. Accessed: 2024-07-16.

[3] Meta Quest 3. https://www.meta.com/ca/quest/quest-3/?srsltid=AfmBOoou9apPeAuq-6ql2xxhkqWrVu1HCDXon_bunlHpwUth9IjZdJTl. Accessed: 2024-08-14.

[4] Meta Quest 3S. https://www.meta.com/ca/quest/quest-3s/?srsltid=AfmBOooqIomZlzGYDjqQqBtBSawQBk9m2GhoVuL Accessed: 2024-08-14.

[5] Meta Developer Documentation. https://developers.facebook.com/docs/. Accessed: 2024-08-14.

[6] Andrade, S.A., Nunes, F.L.S., Delamaro, M.E.: Towards the systematic testing of virtual reality programs. In: 2019 21st symposium on virtual and augmented reality (SVR), pp 196 205 (2019). https://doi.org/10.1109/SVR.2019.00044

[7] Doerner, R., Broll, W., Grimm, P., et al.: (eds) Virtual and Augmented Reality (VR/AR): Foundations and Methods of Extended Realities (XR). Springer Intl. Publishing (2022). https://doi.org/10.1007/978-3-030-79062-2

[8] Gu, R., Rojas, J. M., Shin, D. (2025). Software testing for extended reality applications: a systematic mapping study. Automated Software Engineering, 32(2). https://doi.org/10.1007/s10515-025-00523-7

[9] Beizer, B. (1995). Black-Box testing: Techniques for Functional Testing of Software and Systems. Wiley.

[10] Nidhra, S. (2012). Black Box and White box testing Techniques - A literature review. International Journal of Embedded Systems and Applications, 2(2), 29–50. https://doi.org/10.5121/ijesa.2012.2204

[11] Verma, A., Khatana, A., Chaudhary, S. (2017). A comparative study of black box testing and white box testing. International Journal of Computer Sciences and Engineering, 5(12), 301–304. https://doi.org/10.26438/ijcse/v5i12.301304

[12] The art of software testing. (2012). In Wiley eBooks. https://doi.org/10.1002/9781119202486

[13] Ammann, P., Offutt, J. (2016). Introduction to software testing. https://doi.org/10.1017/9781316771273

[14] Levels of testing. (2006). In Springer eBooks (pp. 133–187). https://doi.org/10.1007/0-387-21658-8-6

[15] Baresi, L., Pezzè, M. (2006). An introduction to software testing. Electronic Notes in Theoretical Computer Science, 148(1), 89–111. https://doi.org/10.1016/j.entcs.2005.12.014

[16] Hooda, I., Chhillar, R. S. (2015). Software test process, testing types and techniques. International Journal of Computer Applications, 111(13), 10–14. https://doi.org/10.5120/19597-1433

[17] Arium — An Automation framework for unity. https://medium.com/xrpractices/arium-an-automation-framework-for-unity-xr-d51ed608e8b0. Accessed: 2024-07-20.

[18] Garousi, V., Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. Information and Software Technology, 76, 92–117. https://doi.org/10.1016/j.infsof.2016.04.015

[19] Bons, A., Marín, B., Aho, P., Vos, T. E. (2023). Scripted and scriptless GUI testing for web applications: An industrial case. Information and Software Technology, 158, 107172. https://doi.org/10.1016/j.infsof.2023.107172

[20] Leotta, M., Clerissi, D., Ricca, F., Tonella, P. (2013). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. 20th Working Conference on Reverse Engineering (WCRE), 272–281. https://doi.org/10.1109/wcre.2013.6671302

[21] Prasetya, I. S. W. B., Shirzadehhajimahmood, S., Ansari, S. G., Fernandes, P., Prada, R. (2021). An Agent-based Architecture for AI-Enhanced Automated Testing for XR Systems, a Short Paper. 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 213–217. https://doi.org/10.1109/icstw52544.2021.00044

[22] Ricós, F. P. (2022). Scriptless testing for extended reality systems. In Lecture notes in business information processing (pp. 786–794). https://doi.org/10.1007/978-3-031-05760-1-56

[23] Leotta, M., Clerissi, D., Ricca, F., Tonella, P. (2013). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. 20th Working Conference on Reverse Engineering (WCRE), 272–281. https://doi.org/10.1109/wcre.2013.6671302

[24] Arium. https://github.com/thoughtworks/Arium. Accessed: 2024-07-21.

[25] Javerliat, C., Villenave, S., Raimbaud, P., Lavoué, G. (2024). PLUME: Record, replay, analyze and share user behavior in 6DOF XR experiences. IEEE Transactions on Visualization and Computer Graphics, 30(5), 2087–2097. https://doi.org/10.1109/tvcg.2024.3372107

[26] Rodrigues, E. M., Saad, R. S., Oliveira, F. M., Costa, L. T., Bernardino, M., Zorzo, A. F. (2014). Evaluating capture and replay and model-based performance testing tools. ESEM, 1–8. https://doi.org/10.1145/2652524.2652587

[27] Liu, C. H., Lu, C. Y., Cheng, S. J., Chang, K. Y., Hsiao, Y. C., Chu, W. M. (2014). Capture-Replay testing for Android applications. International Symposium on Computer, Consumer and Control, 1129–1132. https://doi.org/10.1109/is3c.2014.293

[28] Xu, G., Rountev, A., Tang, Y., Qin, F. (2007). Efficient checkpointing of java software using context-sensitive capture and replay. ESEC-FSE '07, 85–94. https://doi.org/10.1145/1287624.1287638

[29] Leotta, M., Clerissi, D., Ricca, F., Tonella, P. (2013b). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. 20th Working Conference on Reverse Engineering (WCRE), 272–281. https://doi.org/10.1109/wcre.2013.6671302

[30] Selenium. https://chromewebstore.google.com/detail/selenium-ide/mooikfkahbdckldjjndioackbalphokd. Accessed: 2024-07-21.

[31] Selenium IDE. https://www.selenium.dev/selenium-ide/. Accessed: 2024-07-21.

[32] Meta Quest 3. https://www.meta.com/ca/quest/quest-3/?srsltid=AfmBOoqsOyf767YJ7CDU4-Fvq4BGWw1c-pGCcuWdy7EO5HlURz-JLHF. Accessed: 2024-07-22.

[33] Unity. https://unity.com. Accessed: 2024-07-22.

[34] XR Interaction Demo Scene in Unity. https://medium.com/@kennethmclachlan11/xr-interaction-demo-scene-in-unity-8ad7218c9c99. Accessed: 2024-07-22.

[35] ChatGPT. https://openai.com/chatgpt/overview/. Accessed: 2024-07-22.

[36] Lv, Z. (2023). Generative artificial intelligence in the metaverse era. Cognitive Robotics, 3, 208–217. https://doi.org/10.1016/j.cogr.2023.06.001

[37] Epstein, Z., Hertzmann, A., Akten, M., Farid, H., Fjeld, J., Frank, M. R., Groh, M., Herman, L., Leach, N., Mahari, R., Pentland, A. "., Russakovsky, O., Schroeder, H., and Smith, A. (2023). Art and the science of generative AI. Science, 380(6650), 1110–1111. https://doi.org/10.1126/science.adh4451

[38] Feuerriegel, S., Hartmann, J., Janiesch, C., and Zschech, P. (2023). Generative AI. Business and Information Systems Engineering, 66(1), 111–126. https://doi.org/10.1007/s12599-023-00834-7

[39] Banh, L., Strobel, G. (2023). Generative artificial intelligence. Electronic Markets, 33(1). https://doi.org/10.1007/s12525-023-00680-1

[40] Jamil, M. A., Arif, M., Abubakar, N. S. A., Ahmad, A. (2016). Software Testing Techniques: A Literature Review. 6th International Conference on Information and Communication Technology for the Muslim World (ICT4M), 177–182. https://doi.org/10.1109/ict4m.2016.045

[41] Prasetya, I. S. W. B., Shirzadehhajimahmood, S., Ansari, S. G., Fernandes, P., Prada, R. (2021b). An Agent-based Architecture for AI-Enhanced Automated Testing for XR Systems, a Short Paper. 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 213–217. https://doi.org/10.1109/icstw52544.2021.00044

[42] Prada, R., Prasetya, I. S. W. B., Kifetew, F., Dignum, F., Vos, T. E. J., Lander, J., Donnart, J., Kazmierowski, A., Davidson, J., Fernandes, P. M. (2020). Agent-based Testing of Extended Reality Systems. 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 414–417. https://doi.org/10.1109/icst46399.2020.00051

[43] Ricós, F. P. (2022b). Scriptless testing for extended reality systems. In Lecture notes in business information processing (pp. 786–794). https://doi.org/10.1007/978-3-031-05760-1-56

[44] Qin, X., Weaver, G. (2024). Utilizing Generative AI for VR Exploration Testing: A Case Study. ASEW '24: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, 228–232. https://doi.org/10.1145/3691621.3694955

[45] Alkhayat, A., Ciranni, B., Tumuluri, R. S., Tulasi, R. S. (2024). Leveraging Large Language Models for Enhanced VR Development: Insights and Challenges. 2024 IEEE Gaming, Entertainment, and Media Conference (GEM), 1–6. https://doi.org/10.1109/gem61861.2024.10585495

[46] Choi, J., Jeong, S., Ko, J. (2022). Emulating Your eXtended World: an emulation environment for XR app development. 2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS), 131–139. https://doi.org/10.1109/mass56207.2022.00025

[47] Qian, X., Wang, T., Xu, X., Jonker, T. R., Todi, K. (2024). Fast-Forward Reality: Authoring Error-Free Context-Aware Policies with Real-Time Unit Tests in Extended Reality. CHI '24: Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, 1–17. https://doi.org/10.1145/3613904.3642158

[48] Research challenges in information Science. (2022). In Lecture notes in business information processing. https://doi.org/10.1007/978-3-031-05760-1

[49] Qin, X., Weaver, G. (2024b). Utilizing Generative AI for VR Exploration Testing: A Case Study. ASEW '24, 228–232. https://doi.org/10.1145/3691621.3694955

[50] Hunko, I., Muliarevych, O., Trishchuk, R., Zybin, S., Halachev, P., National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ukraine, National Academy of Statistics, Accounting and Audit, Ukraine, Computer Engineering Department, Lviv Polytechnic National University, Ukraine, Department of Reprography of the Publishing and Printing Institute, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ukraine, Information Technology Security Department, Faculty of Cybersecurity and Software Engineering, National Aviation University, Ukraine, University of Chemical Technology and Metallurgy, Bulgaria. (2024). THE ROLE OF VIRTUAL REALITY IN IMPROVING SOFTWARE TESTING METHODS AND TOOLS [Journal-article]. Journal of Theoretical and Applied Information Technology, 102(11), 4723–4724. https://www.jatit.org

[51] Paduraru, C., Stefanescu, A., Jianu, A. (2024). Unit Test Generation using Large Language Models for Unity Game Development. FaSE4Games 2024: Proceedings of the 1st ACM International Workshop on Foundations of Applied Software Engineering for Games, 7–13. https://doi.org/10.1145/3663532.3664466

[52] Cheng, Y., Kuo, J. W., Cheng, B., Kuo, C. H. (2015). A Non-intrusive, Platform-Independent Capture/Replay Test Automation System. 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 1122–1127. https://doi.org/10.1109/hpcc-css-icess.2015.138

[53] Liu, C. H., Lu, C. Y., Cheng, S. J., Chang, K. Y., Hsiao, Y. C., Chu, W. M. (2014). Capture-Replay testing for Android applications. International Symposium on Computer, Consumer and Control, 1129–1132. https://doi.org/10.1109/is3c.2014.293

[54] Wang, X. (2022). VRTest. 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). https://doi.org/10.1145/3510454.3516870

[55] Chen, W., Liu, C., Chen, P., Hsu, C. (2015). A game framework supporting automatic functional testing for games. In Studies in computational intelligence (pp. 13–26). https://doi.org/10.1007/978-3-319-26396-0-2

[56] Xie, X., Liu, H., Zhang, Z., Qiu, Y., Gao, F., Qi, S., Zhu, Y., Zhu, S. (2019). VRGym. Proceedings of the ACM Turing Celebration Conference - China, 1–6. https://doi.org/10.1145/3321408.3322633

[57] Brandstätter, K., Steed, A. (2023). Dialogues For One: Single-User Content Creation Using Immersive Record and Replay. VRST 2023, 36, 1–11. https://doi.org/10.1145/3611659.3615695

[58] Donvir, A., Panyam, S., Paliwal, G., Gujar, P. (2024). The Role of Generative AI Tools in Application Development: A Comprehensive Review of Current Technologies and Practices. 2024 International Conference on Engineering Management of Communication and Technology (EMCTECH), 1–9. https://doi.org/10.1109/emctech63049.2024.10741797

[59] C. Ebert and P. Louridas, "Generative AI for Software Practitioners," in IEEE Software, vol. 40, no. 4, pp. 30-38, July-Aug. 2023, doi: 10.1109/MS.2023.3265877.

[60] Bengesi, S., El-Sayed, H., Sarker, M. K., Houkpati, Y., Irungu, J., Oladunni, T. (2024). Advancements in Generative AI: A comprehensive review of GANs, GPT, autoencoders, diffusion model, and transformers. IEEE Access, 12, 69812–69837. https://doi.org/10.1109/access.2024.3397775

[61] Sun, J., Liao, Q. V., Muller, M., Agarwal, M., Houde, S., Talamadupula, K., Weisz, J. D. (2022). Investigating Explainability of Generative AI for Code through Scenario-based Design. IUI '22. https://doi.org/10.1145/3490099.3511119

[62] YAML Tutorial: A Complete Guide to Language, Format, and Syntax. https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started. Accessed: 2024-07-24.

[63] Getting started with 3D Automation: Arium at glance. https://medium.com/xrpractices/getting-started-with-3d-automation-arium-at-glance-fca27273426d. Accessed: 2024-07-24.

[64] The Scene View - Unity Official Tutorials. https://www.youtube.com/watch?v=nG0fXdXylMI. Accessed: 2024-07-25.

[65] OpenAI o3-mini. https://openai.com/index/openai-o3-mini/. Accessed: 2024-07-25.

[66] GameDriver. https://gamedriver.io/. Accessed: 2024-07-25.

[67] Brooke, J. (1996). SUS: a "Quick and Dirty" usability scale. In CRC Press eBooks (pp. 207–212). https://doi.org/10.1201/9781498710411-35

[68] L. du Bousquet and N. Zuanon, "An overview of lutess: A specification-based tool for testing synchronous software," in Proceedings of the 14th Conference on Automated Software Engineering (ASE'99), 1999, pp. 208–215.

[69] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical observations on software testing automation," in Proceedings of the Second International Conference on Software Testing Verification and Validation (ICST 2009), 2009, pp. 201–209.

[70] Lobato, L. L., Da Mota Silveira Neto, P., Machado, I. D. C. (2012). A study on risk management for software engineering. 16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012), 47–51. https://doi.org/10.1049/ic.2012.0006

[71] Gu, R., Rojas, J. M.,Shin, D. (2025). Software testing for extended reality applications: a systematic mapping study. Automated Software Engineering, 32(2). https://doi.org/10.1007/s10515-025-00523-7

[72] Moxley, R. T. (1990). Functional testing. Muscle and Nerve, 13(S1), S26–S29. https://doi.org/10.1002/mus.880131309

[73] Afzal, W., Torkar, R., Feldt, R. (2009). A systematic review of search-based testing for non-functional system properties. Information and Software Technology, 51(6), 957–976. https://doi.org/10.1016/j.infsof.2008.12.005

[74] Bartram, D., Bayliss, R. (1984). Automated testing: Past, present and future. Journal of Occupational Psychology, 57(3), 221–237. https://doi.org/10.1111/j.2044-8325.1984.tb00164.x

[75] Berner, S., Weber, R., Keller, R. K. (2005). Observations and lessons learned from automated testing. ICSE '05: Proceedings of the 27th International Conference on Software Engineering, 571. https://doi.org/10.1145/1062455.1062556

[76] 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (COMPSACW 2012): Izmir, Turkey, 16 - 20 July 2012. (2012).

[77] Khan, R., Srivastava, A. K., Pandey, D. (2016). Agile approach for Software Testing process. Conference: 2016 International Conference System Modeling Advancement in Research Trends (SMART), 3–6. https://doi.org/10.1109/sysmart.2016.7894479

[78] Jamil, M. A., Arif, M., Abubakar, N. S. A., Ahmad, A. (2016). Software Testing Techniques: A Literature Review. 2016 6th International Conference on Information and Communication Technology for the Muslim World (ICT4M), 177–182. https://doi.org/10.1109/ict4m.2016.045

[79] Mohialden, Y. M., Hussien, N. M., Hameed, S. A. (2022). Review of software testing methods. Journal La Multiapp, 3(3), 104–112. https://doi.org/10.37899/journallamultiapp.v3i3.648

[80] Alqahtani, H., Kavakli-Thorne, M., Kumar, G. (2019). Applications of Generative Adversarial Networks (GANs): An updated review. Archives of Computational Methods in Engineering, 28(2), 525–552. https://doi.org/10.1007/s11831-019-09388-y

[81] Asesh, A. (2023). Variational Autoencoder Frameworks in Generative AI Model. 2023 24th International Arab Conference on Information Technology (ACIT), 01–06. https://doi.org/10.1109/acit58888.2023.10453782

[82] What Is a Transformer Model?. https://blogs.nvidia.com/blog/what-is-a-transformer-model/. Accessed: 2024-08-15.

[83] What is model training?. https://www.ibm.com/think/topics/model-training. Accessed: 2024-08-15.

[84] Feng, Y., Vanam, S., Cherukupally, M., Zheng, W., Qiu, M., Chen, H. (2023). Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data. 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC), 876–885. https://doi.org/10.1109/compsac57700.2023.00117

[85] Liu, C., Bao, X., Zhang, H., Zhang, N., Hu, H., Zhang, X., Yan, M. (2024). Guiding ChatGPT for Better code Generation: An Empirical study. 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 102–113. https://doi.org/10.1109/saner60148.2024.00018

[86] SWEbench. https://www.swebench.com. Accessed: 2024-09-3.

[87] Tukey, J. W. (1984). The Collected Works of John W. Tukey: More mathematical, 1938-1984.

Appendix A

Pre-Study Survey

Pre-Study Questionnaire

* Required

1. How would you rate your XR development skills using Unity? (1: Not very familiar – I only know a little; 10: I'm a professional developer) *

1	2	3	4	5	6	7	8	9	10

2. How would you rate your Software testing skills using Unity? (1: Not very familiar – I only know a little; 10: I'm a professional tester) *

1	2	3	4	5	6	7	8	9	10

3. Are you familiar with the Arium testing framework for Unity? *

- ◯ Yes
- ◯ No

4. Have you ever used generative AI (such as ChatGPT) for coding? *

- ◯ Yes
- ◯ No

5. Rate your experience with Capture and Replay testing method (1: Not very familiar – I only know a little; 10: I'm a professional tester). *

1	2	3	4	5	6	7	8	9	10

6. If you have experience with capture and reply testing method, please mention it here.

Appendix B

Post Study Surveys

SUS

* Required

1. I think that could use this system frequently *

1	2	3	4	5

2. I found the system unnecessarily complex. *

1	2	3	4	5

3. I thought the system was easy to use. *

1	2	3	4	5

4. I think that I would need the support of a technical person to be able to use this system. *

1	2	3	4	5

5. I found the various functions in this system were well integrated. *

1	2	3	4	5

6. I thought there was too much inconsistency in this system. *

1	2	3	4	5

7. I would imagine that most people would learn to use this system very quickly. *

1	2	3	4	5

8. I found the system very cumbersome to use. *

1	2	3	4	5

9. I felt very confident using the system. *

1	2	3	4	5

10. I needed to learn a lot of things before I could get going with this system. *

1	2	3	4	5

Task Load

1-- Low Demand 10-- Very High Demand

* Required

1. **Mental Demand:** *How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?* *

1	2	3	4	5	6	7	8	9	10

2. **Physical Demand:** *How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?* *

1	2	3	4	5	6	7	8	9	10

3. **Temporal Demand:** *How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?* *

1	2	3	4	5	6	7	8	9	10

4. **Performance:** *How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?* *

1	2	3	4	5	6	7	8	9	10

5. **Effort:** *How hard did you have to work (mentally and physically) to accomplish your level of performance?* *

1	2	3	4	5	6	7	8	9	10

6. **Frustration Level:** *How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?* *

1	2	3	4	5	6	7	8	9	10

Appendix C

Raw Data

Participant	XR Skill	Testing Skill	Arium Experience	GenAI Experience
P1	7	4	No	Yes
P2	7	6	No	Yes
P3	7	7	Yes	Yes
P4	6	4	No	Yes
P5	8	6	Yes	Yes
P6	7	4	No	Yes
P7	6	7	No	Yes
P8	7	5	No	Yes
P9	6	7	No	Yes
P10	8	3	No	Yes
P11	7	6	No	Yes
P12	7	7	No	Yes
P13	6	6	No	Yes
P14	8	2	No	Yes
P15	2	2	No	Yes

Table C.1: Participants' Skills in XR, Testing, Arium, and Generative AI

Participant	Capture & Replay Experience	Domain/Notes
P1	1	–
P2	1	–
P3	1	–
P4	5	Web Testing
P5	1	–
P6	1	–
P7	1	Web Testing
P8	1	–
P9	6	–
P10	1	Web Testing
P11	5	–
P12	1	–
P13	1	–
P14	1	Web Testing
P15	1	–

Table C.2: Participants' Capture & Replay Experience and Domain Expertise

Participant	TestGPT	Manual Coding	Capture and Replay	Comment
P1	20 min – 5 tests	20 min – 3 tests	12 min – 6 tests	–
P2	20 min – 5 tests	20 min – 3 tests	8 min – 6 tests	Needed to ask follow-up questions during TestGPT use
P3	20 min – 3 tests	20 min – 2 tests	10 min – 6 tests	–
P4	20 min – 6 tests	20 min – 5 tests	8 min – 6 tests	–
P5	20 min – 3 tests	17 min – 6 tests	6 min – 6 tests	–
P6	20 min – 4 tests	20 min – 2 tests	8 min – 6 tests	–
P7	20 min – 6 tests	20 min – 5 tests	6 min – 6 tests	–
P8	20 min – 5 tests	18 min – 6 tests	8 min – 6 tests	–
P9	20 min – 6 tests	20 min – 1 test	8 min – 6 tests	Difficulty using Arium for the first time
P10	14 min – 6 tests	20 min – 3 tests	10 min – 6 tests	–
P11	18 min – 6 tests	16 min – 6 tests	6 min – 6 tests	–
P12	20 min – 6 tests	20 min – 3 tests	8 min – 6 tests	–
P13	20 min – 6 tests	20 min – 4 tests	6 min – 6 tests	–
P14	10 min – 6 tests	15 min – 6 tests	8 min – 6 tests	–
P15	12 min – 6 tests	20 min – 1 test	6 min – 6 tests	Difficulty using Arium for the first time

Table C.3: Results of the Study

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Q1	3	4	2	4	5	1	2	4	4	3	4	4	1	3	4
Q2	3	1	2	1	3	2	2	2	2	4	1	2	5	3	1
Q3	3	4	2	4	4	3	4	3	3	4	4	4	1	4	4
Q4	3	1	1	4	4	1	1	1	2	2	3	1	3	2	1
Q5	3	5	2	4	3	1	5	5	3	4	4	4	1	4	4
Q6	1	1	1	2	1	4	1	1	3	2	1	1	2	3	2
Q7	1	4	1	5	3	2	5	4	2	3	4	5	1	4	4
Q8	4	1	4	4	2	4	1	4	3	3	3	1	1	4	1
Q9	3	5	3	4	4	5	4	5	3	3	5	3	1	3	4
Q10	5	1	4	3	4	2	2	3	4	4	2	3	5	2	2

Table C.4: SUS Questionnaire Responses for Arium by Participant

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Q1	2	5	5	4	5	4	2	4	5	5	3	5	4	5	4
Q2	3	2	1	3	3	1	1	1	1	3	2	1	1	1	2
Q3	2	4	5	4	4	5	3	4	5	4	4	4	5	5	4
Q4	1	1	1	4	2	1	1	1	1	4	3	1	1	1	2
Q5	2	4	5	5	3	4	4	4	3	2	4	2	5	5	4
Q6	4	2	1	2	2	4	1	1	3	4	2	1	1	1	2
Q7	3	4	5	3	4	5	4	5	5	4	4	5	5	5	5
Q8	4	1	1	5	2	2	1	5	1	3	3	1	5	1	1
Q9	4	3	5	5	3	4	3	5	5	2	3	3	5	5	4
Q10	2	2	1	3	1	1	2	1	2	3	3	3	1	1	1

Table C.5: SUS Questionnaire Responses for TestGPT by Participant

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Q1	5	5	4	4	2	5	3	4	5	5	4	5	5	4	3
Q2	3	1	1	1	3	3	1	1	1	1	1	1	1	1	1
Q3	4	4	5	5	3	1	4	2	5	5	5	5	5	5	4
Q4	1	1	1	1	4	1	1	1	1	2	2	2	2	1	1
Q5	3	5	3	5	3	5	4	4	5	3	4	5	5	5	3
Q6	2	1	1	1	2	1	1	1	1	3	3	1	1	1	2
Q7	4	4	5	5	3	5	4	5	5	4	5	5	5	5	5
Q8	2	1	1	5	3	2	1	5	1	1	3	1	5	1	2
Q9	4	4	5	5	3	5	5	4	5	5	5	4	5	4	2
Q10	2	1	1	2	4	1	2	1	1	1	2	4	1	1	1

Table C.6: SUS Questionnaire Responses for Capture and Replay by Participant

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Q1	8	3	8	6	5	6	7	4	8	5	2	3	9	4	6
Q2	3	1	1	8	1	6	1	1	1	2	5	2	7	1	1
Q3	7	5	6	4	4	8	4	1	8	4	5	2	10	1	3
Q4	6	4	7	7	10	4	8	9	5	6	9	5	4	8	5
Q5	8	3	7	5	5	5	3	1	8	5	5	3	8	3	7
Q6	3	3	3	4	2	6	2	1	6	6	3	3	10	1	2

Table C.7: NASA-TLX Responses for Arium by Participant

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Q1	5	2	4	6	3	3	6	5	3	4	4	2	2	1	2
Q2	2	1	1	7	1	1	1	1	1	1	2	1	1	1	1
Q3	5	5	4	3	6	6	3	1	2	3	2	1	2	1	1
Q4	5	7	9	7	8	10	3	8	8	7	3	9	7	10	7
Q5	5	3	4	5	4	4	3	3	3	1	4	2	2	1	2
Q6	4	3	2	3	2	1	2	3	1	5	6	1	1	1	1

Table C.8: NASA-TLX Responses for TestGPT by Participant

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Q1	6	1	3	2	6	2	2	4	2	2	2	1	1	3	1
Q2	4	2	1	4	1	4	1	1	1	1	1	1	1	1	1
Q3	5	2	2	1	1	1	1	3	1	1	2	1	1	1	1
Q4	8	10	10	10	10	10	10	10	9	3	9	9	10	10	9
Q5	4	3	2	2	1	1	2	1	2	1	4	1	2	2	1
Q6	2	3	1	1	1	2	1	1	1	1	3	1	1	1	1

Table C.9: NASA-TLX Responses for Capture and Replay by Participant

Participant	Simple Project	Complicated Project
P1	TestGPT	Manual Coding
P2	Capture and Replay	TestGPT
P3	TestGPT	Manual Coding
P4	Capture and Replay	TestGPT
P5	TestGPT	Manual Coding
P6	TestGPT	TestGPT
P7	Capture and Replay	TestGPT
P8	Capture and Replay	TestGPT
P9	TestGPT	TestGPT
P10	Capture and Replay	Capture and Replay
P11	TestGPT	TestGPT
P12	Capture and Replay	TestGPT
P13	TestGPT	TestGPT
P14	Capture and Replay	Capture and Replay
P15	TestGPT	TestGPT

Table C.10: Preferred Testing Method by Participant for Simple and Complicated Projects

Appendix D

Ethical Approval

Conjoint Faculties Research Ethics Board
Research Services Office
2500 University Drive, NW
Calgary AB T2N 1N4
Telephone: (403) 220-4283/6289
cfreb@ucalgary.ca

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

The Conjoint Faculties Research Ethics Board (CFREB), University of Calgary has reviewed and approved the below research. The CFREB is constituted and operates in accordance with the current version of the *Tri-Council Policy Statement: Ethical Conduct for Research Involving Humans* (TCPS).

Ethics ID:	REB25-0214
Principal Investigator:	Frank Maurer
Co-Investigator(s):	
Student Co-Investigator(s):	Seyed Nami Modarressi
Study Title:	Comparing LLM-Generated Code, Manual Testing, and Capture-and-Replay Methods for XR Applications
Sponsor:	

Effective: 11-Apr-2025 **Expires:** 11-Apr-2026

The following documents have been approved for use:

- Recruitment Notice for User Study, February 20, 2025
- Study Consent Form, February 20, 2025
- Pre-Study Questionnaire, February 25, 2025
- Task Load, February 20, 2025
- SUS, February 20, 2025

Restrictions:

This Certification is subject to the following conditions:

1. Approval is granted only for the research and purposes described in the application.
2. Any modification to the approved research must be submitted to the CFREB for approval.
3. Reportable events (SAE's, new safety information, protocol deviations, audit findings, privacy breaches, and participant complaints) are to be submitted in accordance with the Board's reporting requirements.
4. An annual application for renewal of ethics certification must be submitted and approved by the above expiry date.

5. A closure request must be sent to the CFREB when the research is complete or terminated.

Approval by the REB does not necessarily constitute authorization to initiate the conduct of this research. The Principal Investigator is responsible for ensuring required approvals from other involved organizations (e.g., Alberta Health Services, community organizations, school boards) are obtained.

Approved By:	**Date:**
Adam McCrimmon, Ph.D., Chair, CFREB	11-Apr-2025 3:58 PM

Note: This correspondence includes an electronic signature (validation and approval via an online system).

Appendix E

Study Consent Form

UNIVERSITY OF CALGARY

Name of Researcher, Faculty, Department, Telephone & Email:

Nami Modarressi – MSc Student
Department of Computer Science
E-mail: seyednami.modarressi@ucalgary.ca
Phone: (368)-997-5330

Supervisor:

Frank Maurer – Professor
Department of Computer Science
E-mail: frank.maurer@ucalgary.ca
Phone: (403) 220-3531

Title of Project:

Comparing LLM-Generated Code, Manual Testing, and Capture-and-Replay Methods for XR Applications

This consent form, a copy of which has been given to you, is only part of the process of informed consent. If you want more details about something mentioned here, or information not included here, you should feel free to ask. Please take the time to read this carefully and to understand any accompanying information.

The University of Calgary Conjoint Faculties Research Ethics Board has approved this study (REB25-0214).

Purpose of the Study:

The primary purpose of the study is to answer the research question "How does software testing using generative AI compare to traditional manual testing and capture-and-replay methods for XR applications"

What Will I Be Asked To Do?

If you agree to participate in this study, you will be asked to participate in the following different research activities:

1. You will be asked to complete a pre-study questionnaire to assess your familiarity with head-mounted devices, software testing, and XR development. This questionnaire should take approximately 5 minutes.

2. You will be asked to review the documentation provided before each step to become familiar with the testing tool required for that step.

3. You will be asked to read the description of each step and test the provided test scenario using the tool designated for that phase.

4. A researcher will observe, take notes, and videotape your interactions and responses until you feel satisfied with your answers. You may also be asked to complete a post-study questionnaire about your experience after each task. Completing each round should take approximately 20 minutes, and filling out the post-study questionnaires should take about 5 minutes per round.

Participation is voluntary, and you may withdraw at any time. The entire process should last no longer than approximately 90 minutes (three 30-minute steps).

What Type of Personal Information Will Be Collected?

The experiment session includes the interactions with the system and the interview will be audio and video recorded. If you agree to participate, we will ask you to write down your comments during the study. Information such as your Experience with XR development, Experience with software testing may be recorded. In all written publications and presentations based on this research, you will remain anonymous and your comments from the interviews will be referred to with either a participant number or a pseudonym. However, you have until 1 month after participating in the study to ask for withdrawal of your data from the study.

There are several options for you to consider if you decide to take part in this research. We will conduct the experiment only after your acceptance of audio and video recordings. You can choose all, some or none of them. Please put a check mark on the corresponding line(s) that grants us your permission to:

I grant permission to save questionnaire data:	Yes: ___ No: ___
I grant permission to be audio taped:	Yes: ___ No: ___
I grant permission to be video taped:	Yes: ___ No: ___

Are there Risks or Benefits if I Participate?

Your benefits include: Experiencing Virtual Reality. Wearing Head Mounted Devices has potential side effects such as nausea and eye fatigue. The risk of having these side effects is low. There is a rare risk immersion could incite seizure symptoms for the first time. However, we will terminate the study session if you feel uncomfortable.

What Happens to the Information I Provide?

Participation is completely voluntary. You are free to discontinue participation at any time during the study. All the information would be stored in Secure Computing Platform provided by University of Calgary. If you choose to withdraw from the study, any and all information you have provided will be destroyed. No one except the researchers and their supervisors will be allowed to see or hear any of the answers to the questionnaire or task performance. There are no names on the questionnaires. All the collected data will be kept by the investigators indefinitely, and it might be used, in anonymized form, with publications and thesis purposes.

Signatures (written consent)

Your signature on this form indicates that you 1) understand to your satisfaction the information provided to you about your participation in this research project, and 2) agree to participate as a research subject.

In no way does this waive your legal rights nor release the investigators, sponsors, or involved institutions from their legal and professional responsibilities. You are free to withdraw from this research project at any time. You should feel free to ask for clarification or new information throughout your participation.

Participant's Name: (please print) _____

Participant's Signature _____Date: _____

Researcher's Name: (please print) _____

Researcher's Signature: _____Date: _____

Questions/Concerns

If you have any further questions or want clarification regarding this research and/or your participation, please contact:

Seyed Nami Modarressi
Department of Computer Science
(368)997-5330 ,seyednami.modarressi@Ucalgary.ca,

Or
Frank Maurer
Department of Computer Science
(403) 220-3531, frank.maurer@ucalgary.ca

If you have any concerns about the way you have been treated as a participant, please contact the Research Ethics Analyst, Research Services, University of Calgary at (403) 220-4283 or (403) 220-8640; e-mail cfreb@ucalgary.ca.

A copy of this consent form has been given to you to keep for your records and reference. The investigator has kept a copy of the consent form.