ARCHIE++ : A Cloud-enabled Framework for Conducting AR System Testing in the Wild

Sarah M. Lehman, Semir Elezovikj, Haibin Ling, and Chiu C. Tan

Abstract—In this paper, we present ARCHIE++, a testing framework for conducting AR system testing and collecting user feedback in the wild. Our system addresses challenges in AR testing practices by aggregating usability feedback data (collected *in situ*) with system performance data from that same time period. These data packets can then be leveraged to identify edge cases encountered by testers during unconstrained usage scenarios. We begin by presenting a set of current trends in performing human testing of AR systems, identified by reviewing a selection of recent work from leading conferences in mixed reality, human factors, and mobile and pervasive systems. From the trends, we identify a set of challenges to be faced when attempting to adopt these practices to testing in the wild. These challenges are used to inform the design of our framework, which provides a cloud-enabled and device-agnostic way for AR systems developers to improve their knowledge of environmental conditions and to support scalability and reproducibility when testing in the wild. We then present a series of case studies demonstrating how ARCHIE++ can be used to support a range of AR testing scenarios, and demonstrate the limited overhead of the framework through a series of evaluations. We close with additional discussion on the design and utility of ARCHIE++ under various edge conditions.

Index Terms—Augmented reality, testing and debugging, mobile applications, human-centered computing

1 Introduction

Augmented reality (AR) systems are ones which leverage knowledge about the environment to generate and integrate virtual content into the user's experience of the real world. These systems are increasing in popularity and prevalence with applications in domains such as manufacturing [24], [37], healthcare [15], [47], education [12], [20], retail [17], [22], and beyond. AR applications for mobile devices such as smartphones and tablets are some of the most popular; indeed, the AR social media application Snapchat [27] has been downloaded and installed over 1 billion times from the Google Play marketplace.

As mobile AR systems become more commonplace, there is a corresponding need for tools to help test them. Such systems are comprised of many different technological components, such as computer vision and machine learning modules, the user interface implementation, handling of sensor data streams, and any additional boilerplate logic needed just to run on a particular device (e.g. Android, iOS, etc.). There are many places for bugs to appear, both within and between components. While testing in the lab is an effective first step in identifying and "squashing" bugs, the overly controlled nature of laboratory test conditions do not present a realistic view of system performance and usability. Eventually developers will want to transition to testing *in the wild*, that is, giving the application to testers to use in their day-to-day lives. This, however, introduces even more variables, as testers are now unconstrained in their environments and interactions with the app. Certain application components, such as computer vision and machine learning modules, can be notoriously sensitive to unconstrained conditions as environmental factors such as ambient light levels, user movement speed, viewing angle, level of background processing, and many other factors can have direct impacts on the module's ability perform its task. These are known as *edge cases*, that is, scenarios in which unanticipated or extreme system inputs yield undesirable outputs. Edge cases can be particularly difficult to detect, as the cost for exhaustively testing an AR system will every possible input can be prohibitively high.

In addition to system performance and edge case testing, there is a human aspect which makes mobile AR system testing unique. Because the virtual content generated by the system is integrated directly into the user's experience of the real world, incorrectly chosen or poorly placed and formatted virtual content can have very real impacts on the user. Many AR system users experience "simulator sickness", or nausea and disorientation caused by a disparity between the appearance and movement of virtual content compared to how the user expects real-world objects to move [32]. This is particularly prevalent with testers using head-mounted displays (HMDs) such as the Microsoft HoloLens [26], [48]. Even in handheld systems, virtual content in mobile AR apps has the potential to obscure and distract from important real-world content, such as street signs or approaching cars when playing Pokemon Go, which has significant impacts to user safety [14], [29].

These sorts of issues are typically identified through *usability testing*, where human testers are presented with the system, asked to perform a set task, and then asked for feedback on how well the system performed for the assigned use case. Traditional methods of feedback collection are limited, however, in that, while they can be reliable when reporting *satisfactory* system use, they fail to capture any

- *Ms. Lehman, Mr. Elezovikj, and Dr. Tan are with Temple University, Philadelphia PA 19122.* ***Email**: {smlehman, semir, cctan}@temple.edu*
- *Dr. Ling is with Stony Brook University, Stony Brook NY 11794.* ***Email**: hling@cs.stonybrook.edu*

(a) (b) (c)

Fig. 1: Screenshots of ARCHIE++ in use; showcasing prototype comparing (a) small and (b) large text sizes for object labels, and (c) collecting feedback with form overlaid on top of app UI

TABLE 1: Summary of functional additions when using ARCHIE++ vs. the original system [33]

	ARCHIE 1.0	ARCHIE++
Configurable multi-scenario testing	X	X
In-situ user feedback collection	X	X
Real-time system state data collection	X	X
Distributed artifact storage		X
Configurable post-eval. processing		X
Scalable for large test groups		X
Device-agnostic		X

quantitative, contextual information about scenarios which precipitate *poor* feedback (that is, when the tester encounters an edge case). For instance, a tester evaluating a shopping assistant application may be able to tell the research team that the labels placed over items in the store were hard to read, but may have trouble recounting additional specifics. This problem is exacerbated when testers are evaluating the system over long periods of time, or experience a gap between system use and feedback collection, and so must rely on their memories to describe such events.

To help address these needs, we present **ARCHIE++, the Augmented Reality Computer-Human Interaction Evaluator framework**, a runtime performance and usability data collection system which facilitates system testing and edge case detection for mobile AR systems in the wild. This framework builds on prior work [33], and makes the following new contributions:

- **Identified trends and challenges when conducting AR system testing.** We have conducted a study of recent work that presents and performs usability testing on AR systems for a range of problem spaces. Through this study, we identified three challenges that researchers face when testing mobile AR systems in the wild: incomplete knowledge of test conditions, scalability, and reproducibility.
- **Developed ARCHIE++, a cloud-enabled, device-agnostic framework for conducting in-the-wild testing of mobile AR systems.** To address the challenges highlighted by our motivating study, we developed a scalable framework to assist researchers in conducting AR system testing. We implemented ARCHIE++ (shown in Figure 1) as a library plugin for Unity3D [44], a popular development environment for mixed reality experiences which supports a range of platforms (e.g. Android, iOS, HoloLens, and more). This library is supported by a Firebase backend [25] for data aggregation and processing. A summary of functional improvements of ARCHIE++ compared to the original ARCHIE 1.0 framework are summarized in Table 1.
- **Demonstrated how ARCHIE++ can be used in practice using a range of common AR scenarios.** Finally, we present three case studies representing common scenarios within AR research, to demonstrate the utility and versatility of our framework. In doing so, we highlight three primary features offered by ARCHIE++: comparison testing, runtime diagnostics, and long-term testing.

The rest of the paper is organized as follows: Section 2 discusses the related work, while Section 3 examines usability testing practices in recent AR research efforts, and identifies subsequent challenges when attempting to conduct testing in the wild. Section 4 presents an architectural overview of the ARCHIE++ framework, and describes the phases involved in its use. Sections 5 and 6 present, respectively, a set of case studies describing how the ARCHIE++ framework may be used, as well as our system evaluations. Section 7 provides additional discussion on various design decisions and limitations of our framework, and Section 8 concludes.

2 Related Work

A number of tools and frameworks exist to help developers test their mobile AR systems. These testing platforms serve a range of goals, many of which can be supplemented with ARCHIE++.

Commercial testing tools. The increase in commercially available AR hardware and development environments brings with it an increase in commercial testing tools as well. Oculus provides official guidelines for performance optimization [10] as well as an array of performance monitoring and debugging tools [1], [2], [4], [5] for applications built for the Rift headset. Microsoft provides its own official guidelines for testing applications built for the HoloLens headset [7], in addition to a selection of emulators and simulators [3], [8], [9] for testing applications with pre-determined inputs. Similarly, Apple's latest version of ARKit ships with a testing tool called "Reality Composer" [6], which allows

developers to record and replay system inputs for testing. While all of these tools can be helpful during development and initial testing stages, they are generally suitable only for in-lab testing, as their operations are quite resource-intensive. ARCHIE++ can supplement these efforts by providing additional lightweight support when an application has passed code quality testing in the lab and is ready for user evaluations in the wild.

AR analytics frameworks. There has been some recent work to assist AR researchers in collecting and analyzing data from system evaluations. MRAT [36] enables researchers outside of the Computer Science domain to run complex experiments using AR systems. The framework collects huge amounts of environment and user-specific data to facilitate analysis, but provides no method for testers to provide their own usability feedback. Further, the framework is only viable in the lab, with no support for in-the-wild testing. Another framework described in [40] *does* support testing AR applications in the wild, while collecting a range of system-level metrics about the user base at large as well as individuals. Further, it supports A/B testing, a limited version of comparison testing in which participants are exposed to one of two implementation options, and then feedback is collected; however, the framework gathers this feedback across the user base at large, rather than per individual. Any preference or performance conclusions are drawn based on system metrics analysis rather than individual feedback, which is not collected. Other frameworks, such as the visualization comparison system presented by Brehmer et al [18], use supplementary user behavior, such as periods of inactivity, to preemptively remove incomplete data sets from their test corpus. ARCHIE++ addresses both of these issues by enabling comparison testing *within-subjects* so that every tester experiences all implementations, and providing a configurable post-processing workflow to filter out unsatisfactory data samples.

Usability testing. In addition to more system-level analysis, there has been recent research in developing adaptive systems and testing patterns to improve AR system usability. The popular usability testing survey by Ivory and Hearst [28], though it presents solid guidance on usability feedback collection in general and even touches on comparison testing, was written before the time of handheld mobile and wearable devices. As such, there remains a need for testing tools and procedures that focus on these non-traditional interfaces. The system proposed in [34] dynamically adapts the UI based on both the environment and testers' estimated cognitive load. The drawback to this system is that the adjustments to the UI are only as good as the system's ability to estimate tester discomfort; there is no built-in support for testers to provide *in situ* feedback to modify their experience. Mottelson et. al. [35] present a feasibility study on transitioning virtual reality studies out of the laboratory setting. Their approach, however, focused more on researchers' abilities to recruit and manage participants and to administer a user study remotely rather than to diagnose and understand any issues that might occur with their system during runtime. Costa et. al. [21] present another method for conducting automated user studies, with a standalone application to facilitate studies on participants' information retrieval behavior with online search engines. Similar to [35], Costa et. al. make the simplifying assumption that the system itself is bug-free with a set implementation; they assume that it is only tester behavior and feedback that the researchers are interested in, rather than edge cases within the system itself. Further, the system proposed by Costa et. al. has no out-of-the-box support for comparison testing. Other works explore the usability of AR systems to accomplish a specific task, such as chemistry education [13], [23], assessing the development of motor and cognitive skills in children [39], and workplace training [31], [41]. ARCHIE++ can supplement these testing frameworks by providing grouped packets of system state data and user feedback data, not only to collect usability information in the moment, but also to help developers target and address scenarios that testers explicitly disliked.

3 Understanding AR Testing Practices

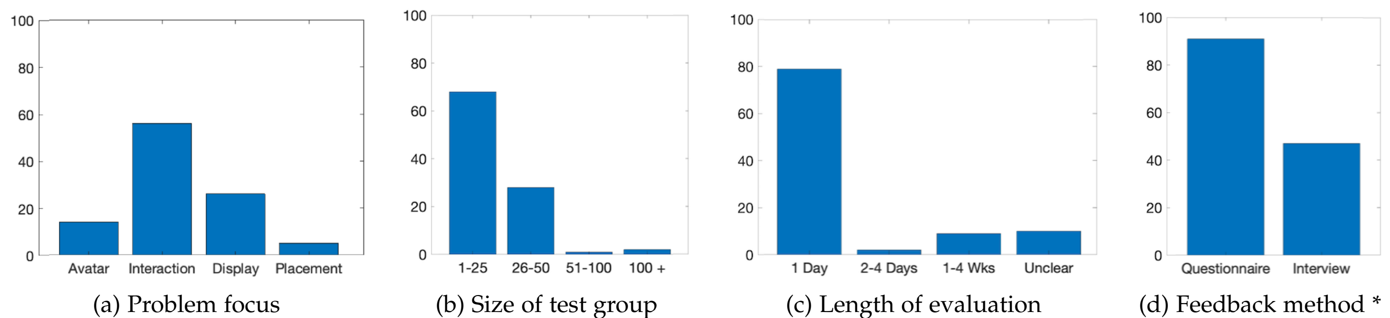
In order to inform our solution, we first conducted a survey of current research efforts in augmented reality with a particular focus on user feedback and testing methods. Our goal was to learn more about how user studies and human testing efforts are conducted, the kinds of information that researchers are seeking to gather from testers, and how that information is collected. This knowledge is useful in that it can help us identify shortcomings in current testing practices that ARCHIE++ can help alleviate.

Comparison with prior work: In [33], we performed an initial investigation to identify broad trends in how AR researchers tested their systems. We then leveraged these trends to identify challenges in migrating existing testing practices to the wild. In this follow-up paper, we expand that initial survey to include results from additional venues. These expanded results confirmed the conclusions from our previous study, and allowed us to identify practical challenges that researchers would face when applying existing practices to large scale testing efforts.

3.1 Methodology

To assemble the body of work for our survey, we reviewed proceedings from six different conferences: two focused on AR and VR (IEEE VR and ISMAR), two focused on human factors (CHI and UIST), and two focused on mobile and pervasive systems (MobiSys and PerCom). These conferences were selected as the dominant venues within a range of AR research disciplines; the goal of which was to provide a more holistic and systems-focused view of user testing procedures within these research efforts than a deeper survey of a single venue could provide on its own. We considered conferences rather than journals because they represent more cutting-edge rather than archival research efforts.

There are many surveys of augmented reality systems available, but very few that address testing and user study techniques within such systems. One survey that *does* touch on this topic is Billinghurst et. al. [16], though it only covers works published through 2014. Even considering only works published from 2015 onward, this represents a substantial body of work (e.g. 370 and 378 Scopus results respectively from IEEE VR and CHI when filtering by augmented or mixed reality). As our interest in this survey was



(a) Problem focus (b) Size of test group (c) Length of evaluation (d) Feedback method *

Fig. 2: User testing trends of surveyed AR systems. Figures marked (*) show papers which fall into multiple categories.

not to present an exhaustive literature review, but rather to gain insights into the latest user study methodologies, we elected to focus on more recent publications in order to yield a more manageable corpus size. For each conference, therefore, we considered the two most recent sets of proceedings spanning the years 2018 to 2020. Papers were included if they mentioned augmented or mixed reality in the titles or abstracts. Papers were excluded if they did not contain a user study, the user study consisted only of qualitative data (such as sensor readings), or the paper primarily focused on non-vision modalities (such as audio or haptics). We also only considered full conference papers; poster, workshop, and other "short" papers were subsequently excluded.

Once the corpus was assembled (N = 81), we categorized each paper as follows. First, we considered the focus of the problem the paper was trying to solve. We assigned each paper to one of four broad categories: avatars, interaction, method of content display, and content placement. We also noted the conditions of the primary user study, namely the environment (whether in the lab or in the wild), the system hardware (HMD, smartphone/tablet, or other), number of testers, evaluation length per tester, and method of feedback collection (such as questionnaires, interviews, or other). Finally, we noted whether the researchers asked testers to explicitly compare or give a preference between multiple UI options.

3.2 Results

From our survey, we observed the following trends, which are summarized in Figure 2.

Problem Focus: More than half (56%) of papers focused on some type of system interaction, typically with regard to the system input method, social interactions, or some facet of task completion. The second most popular focus was on tester perceptions of content display (26%), generally in terms of how realistic or informative a given display method was.

Size, Length of Evaluation: The majority of tests were conducted with fewer than 25 people (68%) with each tester spending only a single day (79%) using the system. Of these instances, each tester typically spent only an hour or two interacting with the application-under-test (AUT). The system equipment being utilized during this time was generally a head-mounted display (65%).

Feedback Collection: The overwhelming majority (91%) of papers used one or more written questionnaires to elicit feedback from testers, supplemented by personal interviews. The most common questionnaires utilized were some variation of the System Usability Scale [46] to judge general system appeal; the NASA Task Load Index [11] questionnaire to measure the cognitive burden associated with completing a task; and some bespoke version of a Mean Opinion Score.

Testing Methodology: More than half (60%) of papers perform *comparison testing*, where they present multiple user experience options and ask their testers to select a preferred condition. This covered a variety of system outputs, from styling of avatars to formatting and placement of text to general system usability, and also included both comparisons against pre-existing systems and varying configurations of the authors' own system.

Test Environment: Of the 81 papers reviewed, only two [30], [38] conducted any sort of unconstrained user study in the wild. This means that 97.5% of papers administered their human testing in laboratory conditions.

3.3 Challenges of Current Testing Practices in the Wild

The results of our survey as described above are unsurprising when considering that such a significant portion of human testing for AR systems is done in heavily constrained and controlled laboratory conditions. This is a direct reflection of the labor-intensive nature of conducting in-lab human testing, which, although an important first step for validating system performance, provides only a limited snapshot into system and user behavior. Testing in the wild, on the other hand, can provide a much more mature and realistic glimpse into how users would actually interact with a system on a daily basis. However, one cannot simply copy lab testing practices directly in the wild and expect success. Below, we describe three different challenges that would have to be addressed in order to transition current AR usability testing practices from the lab to the wild.

Challenge #1 (C1): incomplete knowledge of test conditions. The first challenge is the inability to correlate testers' feedback with system state and other data in order to diagnose the underlying conditions that precipitated specific feedback. Questionnaires and interviews are particularly weak in this area, as they are static snapshots of tester feedback with no relation to the system behavior during the evaluation period that contributed most to the tester's experience. For example, a tester might be able to identify that the system seemed slow and unresponsive, but will not

TABLE 2: Data collected by ARCHIE++ framework. The issue list, raw and augmented screenshots, and FPS trace are bundled into "packets". Packets are labeled with timestamp, config ID, and experience rating before offloading to Firebase.

Collected Item	Data Type	Description
timestamp	Date	Date and time at which interval ended and data was packaged
config_id	String	Identifier of the configuration being tested when data was collected
experience_rating	String	Enumeration reflecting current user experience (e.g. excellent, good, fair, poor, bad)
issue_list	String[]	Configurable list of issues encountered by the user during this interval (e.g. poor contrast, etc.)
screenshot_raw	Image	Unaltered frame taken from device camera feed
screenshot_aug	Image	Altered camera frame, containing application-generated augmentations (e.g. what the user sees)
fps_trace	String[]	History of how many frames were generated per second (FPS) during this interval

be able to indicate whether it was because of factors such as slow processing or poor viewing angles captured by the system camera.

Challenge #2 (C2): scalability. The second challenge is in the difficulty in scaling current testing practices. Current human testing practices focus on working with small groups of people for short periods of time. This is because testers' time is valuable, and recruitment of testers that meet desired criteria may be difficult. Related to this is the time and labor cost of manually administering questionnaires and interviews to study participants. Further, research teams typically use very expensive or specialized equipment, with bespoke code bases tailored to a specific usage environment. This makes the transition to in-the-wild testing very difficult, as systems may be ill-equipped to support consumer-grade hardware or open-ended usage conditions.

Challenge #3 (C3): reproducibility. The third challenge in contemporary AR testing efforts is the inability to effectively compare and reproduce the testing conditions between test instances. When conducting human testing in the lab, environmental conditions such as ambient light levels, weather, time of day, viewing angles, and movement speed are easily controlled between testers. However, when testing at different locations, or when performing tests in the wild, it is much more difficult to control and reproduce these conditions. Too great of variance between test conditions can subsequently make test results unreliable.

In the following section, we present the design of our ARCHIE++ framework, and demonstrate how it addresses the challenges described here.

4 System Design

When designing ARCHIE++, we improved upon our original design [33] by streamlining the developer-facing framework architecture, and moving data storage and processing functions to the cloud. Developers using ARCHIE++ are responsible only for implementing a single *function call*, *event listener*, and *manifest file* in order to utilize the framework; this means (as shown in Section 6) that modifying an existing code base to incorporate ARCHIE++ requires on average only 97 new lines of code, where integration of ARCHIE 1.0 would require hundreds of new lines of code. In this section, we first describe how the testing challenges from Section 3.3 influenced our system design, followed by a more detailed discussion of the ARCHIE++ framework architecture and its behavior during runtime. The source code for ARCHIE++ is available on GitHub[1].

1. https://github.com/lehmansarahm/ARCHIE

4.1 Testing Challenge Impacts on System Design

For testing of mobile AR applications to work in the wild, the challenges described in Section 3.3 need to be addressed. First, ARCHIE++ addresses the lack of developer understanding of test conditions (C1) by collecting samples of system performance and input data *in tandem with* tester feedback during run-time. The data collected by ARCHIE++ (described in Table 2) is grouped into time-boxed "packets", with the user's "experience rating" (shown in Figure 1c) included in the title. By grouping and labeling data in this way, ARCHIE++ enables researchers to quickly identify and explore contextual information specific to those conditions which precipitated poor user feedback.

Second, ARCHIE++ addresses the problem of scalability (C2) by embracing a device-agnostic cloud-enabled architecture, specifically as a plugin for the popular Unity3D IDE [44] with a Firebase backend [25]. Unity3D is one of the preeminent development environments for AR and VR systems, and includes support for a wide range of devices, including mobile systems, head-mounted displays, and standalone applications for personal computers [43], [45]. Unity even supports browser-based applications, such as the system proposed by Butcher et. al. [19], through integrations with libraries such as WebGL. Thus, by leveraging Unity, developers can build a single application and deploy to a range of tester devices, rather than maintaining individual code bases for each device OS. Similarly, by incorporating Firebase as our framework backend, developers gain infrastructure-as-a-service benefits such as redundancy and high availability, as well as a centralized point of data aggregation and processing. This means that developers can administer evaluations with larger groups for longer periods of time, as they no longer have to meet with testers individually to debrief and retrieve data from their devices.

Finally, ARCHIE++ addresses the problem of reproducibility (C3) by providing camera frames, both in their original state and including application-generated augmentations, to help researchers verify consistency of test conditions under which the framework is being used. Using the raw camera frames, developers can supplement future testing efforts using known problematic inputs previously collected by the system, either by feeding the frames directly back into the system or using the frames as seed values to fuzz and generate new data sets. Subsequent outputs can then be compared against the augmented camera frames to judge performance of the new system. Feedback collected with traditional methods such as questionnaires and interviews are unable to provide this information.

TABLE 3: Settings and properties of ARCHIE++ manifest file

Field	Data Type	Description
config_ids	String[]	List of labels describing the configuration conditions to be tested
config_test_period	Multi-value	Interval during which to test each configuration; made up of value (int) and denomination (string, e.g. "minutes", "hours", etc.)
collect_raw_frame	Bool	Flag indicating whether framework should collect raw (unaugmented) camera frames as part of the system input data collection
collect_aug_frame	Bool	Flag indicating whether framework should collect augmented camera frames as part of the system input data collection

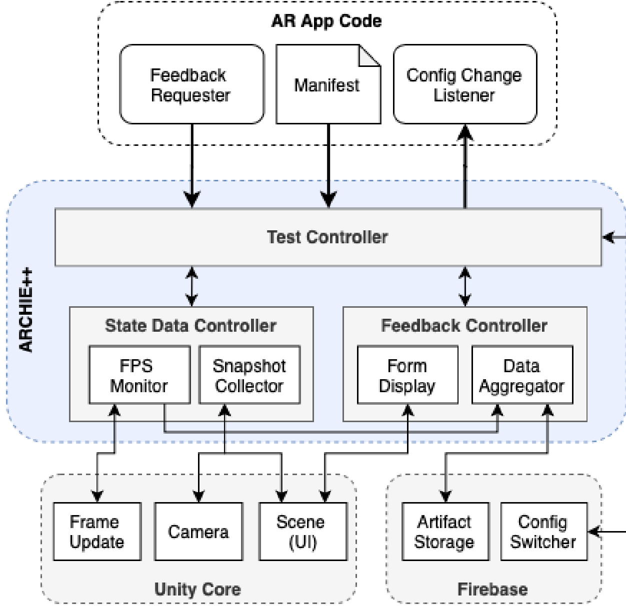


Fig. 3: ARCHIE++ architecture diagram

4.2 Framework Architecture

The core of the ARCHIE++ framework is built on the concept of a **configuration**, which is a particular condition to be evaluated by testers during a given evaluation period. Configurations are completely developer-determined; a configuration could be a particular algorithmic implementation, a UI formatting scheme, or similar set of system characteristics. ARCHIE++ iterates through a collection of configurations during runtime, collecting usability feedback from testers relative to the currently available config, and aggregating that feedback data (along with system input data collected during runtime) in a remote backend.

The system architecture of ARCHIE++ is reflected in Figure 3. The primary component is the plugin for the Unity IDE (shown in blue), which manages the collection of system state and user feedback data. This plugin interfaces with the Unity core library to retrieve camera frames and scene information, as well as calculating and logging the frames displayed per second (FPS). The plugin is also responsible for forwarding data files to Firebase for storage, and raising the appropriate events with the AR application-under-test when it is time for a configuration context switch. The AUT is responsible only for providing a system manifest file to configure the ARCHIE++ test instance, implementing an event listener to respond to configuration changes, and raising the feedback request method as appropriate (shown at topmost level, Figure 3).

Compared to our prior work [33], ARCHIE++ has a greatly simplified client-side architecture. This is due to an observation in our extended survey (described in Section 3) that researchers generally only require testers to evaluate a relatively small number of options (generally two but as many as four). Because of this, *we genericized our handling of configurations, treating them only as event labels*, and allowing developers to respond to a newly selected label in whatever way they want. This allows developers and researchers to use ARCHIE++ to test anything from alternate color schemes for a UI, to competing computer vision modules, to different label placement algorithms, and much more.

Utilization of the ARCHIE++ framework consists of three phases: (1) initialization and pre-deployment, (2) run-time data collection, and (3) post-processing. The following sections describe these phases in more detail.

4.3 Phase 1: Pre-deployment and Initialization

When configuring an application to use ARCHIE++, a developer must first import the Unity plugin and perform a series of one-time steps to integrate Firebase into the application. First, she must create a project for her application within the Firebase developer console. Next, she must download the Firebase SDK and import the base resource package into Unity, as well as the supplementary packages for Authentication, Storage, and Cloud Functions. Finally, she must install the Firebase CLI toolkit to her local machine.

The developer must also provide a **system manifest**, which contains properties controlling the execution of the test instance (described in Table 3). The primary manifest settings include a list of identifiers for the configurations she wants to test, the desired test period (e.g. how long a tester should interact with a given configuration before moving to the next one), and whether to collect raw or augmented frames from the device camera. Allowing the developer to manipulate system parameters directly is an explicit design choice, as it *provides developers the freedom to control framework functionality according to their systems' needs*. For example, the developer might decide that she does not need raw camera frames, and so may configure the framework not to collect them, thus saving transmission and storage bandwidth. A sample system manifest is shown in Section 5.1.

The next step for the developer is to provide an **event listener** to control and respond to the changes in configuration selection. The event listener overrides OnConfigSelected() from the ARCHIE++ TestController class, which is raised by ARCHIE++ when the Firebase back-end selects a new configuration for testing. Allowing the developer to implement her own event listener is also an explicit design choice, as it *eliminates restrictions on what functionality can be tested by users*. For example, one

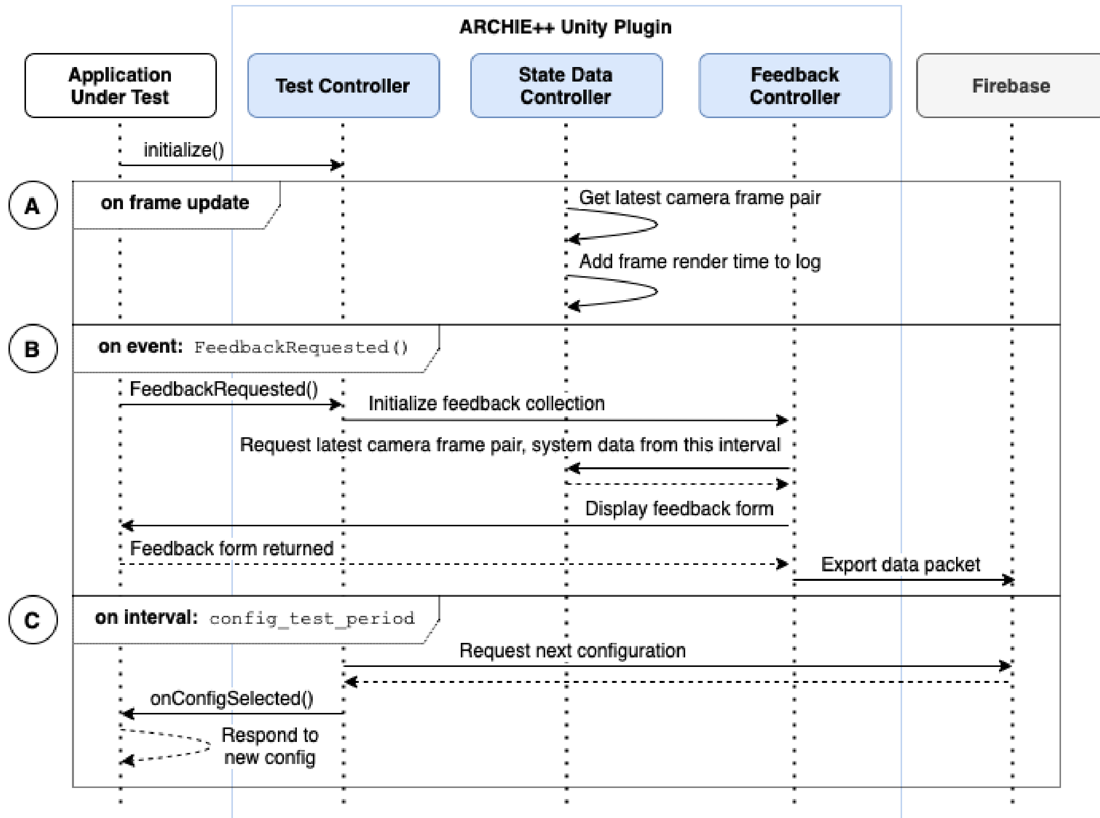


Fig. 4: ARCHIE++ runtime program loops for subordinate processes: (A) on frame update, the State Data Controller keeps a running copy of the last raw/augmented camera frame pair and logs the latest render time; (B) on feedback request from the application-under-test, the Feedback Controller displays the feedback form, aggregates the data packet, and exports it to Firebase; and (C) on `config_test_period`, the Test Controller requests the next configuration from Firebase and raises the `OnConfigSelected()` event with the application-under-test.

developer could declare "light" and "dark" as her configuration IDs, and respond to configuration change events by swapping out illumination algorithms; another could respond by changing the UI color scheme, while yet another could respond by changing an avatar model. It also allows developers to extend the testable functionality at any time by simply expanding a new entry to the list of supported configurations. A sample event listener implementation is presented in Section 5.1.

The final step to incorporating ARCHIE++ is to explicitly call `FeedbackRequested()` from the base TestController class at some point in her code, in order to kick off the feedback collection pipeline. While ARCHIE++ does provide a default feedback form (as shown in Figure 1c), the developer is also free to reconfigure the form at this time, if she is interested in other issues or factors than the preset options. Requiring the developer to explicitly configure the feedback form and make the feedback request is another deliberate design choice, so that the *feedback gathering logic can be triggered at any time that makes sense for the developer and her specific application*. Feedback could be gathered, for example, on an interval, when the user clicks a particular button, when the app recognizes a particular target, a combination of these, or under any other condition. The only requirement is that this method is called explicitly somewhere in the application code.

4.4 Phase 2: Run-time Execution

During run-time, ARCHIE++ acts as a mostly transparent middle layer between the application-under-test and the end user. The run-time logic of the ARCHIE++ framework consists of three primary processes, shown in Figure 4, responsible for assembling and offloading the data packets described in Table 2. The first of these subordinate processes (denoted as A) describes the State Data Controller collecting behavior and input data from the AUT. As Unity refreshes the application UI, the time that it takes to render the scene is appended to a running log (`fps_trace`). In addition, approximately once per second, the raw and augmented camera frames (`screenshot_raw`, `screenshot_aug`) are skimmed from the camera data stream. Only the latest frame pair is maintained; as new frames become available, the old ones are overwritten.

The second subordinate process (B) occurs when the AUT raises the `FeedbackRequested()` event. When this occurs, the Test Controller will initialize feedback collection, grabbing a copy of the system state data from the State Data Controller and displaying the feedback form to the AUT. The feedback form displays on top of the AUT, and includes a general rating for overall system experience (`experience_rating`), as well as a configurable check list of issues that the tester may be experiencing (`issue_list`), such as "Poor Contrast", "Poor Placement", "Wrong Size",

and "Distracting". Once the user has filled out and submitted the feedback form, the Feedback Controller bundles the data packet, labels it with the current `timestamp` and `config_id`, and exports it to Firebase.

The third and final subordinate process (C) demonstrates the Test Controller requesting the next configuration from Firebase after the configured test interval. The interval is determined by the `config_test_period` property of the system manifest file, and can be specified in terms of minutes, hours, or days. Once invoked, the Firebase back-end evaluates the list of configurations, and returns the ID of the next one to test. It should be noted here that, while the current implementation of ARCHIE++ assumes Round Robin scheduling of configurations, the selection logic could easily be expanded to be more strategic, such as the Multi-Arm Bandit approach described in our previous work [33]. The Test Controller then forwards the configuration ID to the AUT, which is able to respond in whatever way is appropriate, such as hot-swapping algorithms or replacing UI components.

4.5 Phase 3: Post-processing and Analysis

With the ability to scale human testing efforts to larger groups of people using ARCHIE++ comes the possibility of having to sort through potentially huge data sets of testing artifacts (or, discrete system by-products) after the evaluation period is over. Therefore, ARCHIE++ includes a post-processing workflow to enable developers to sort through the data collected by the framework, reducing the complexity of the dataset to make the review and debugging process more manageable. Further, this phase provides the opportunity for developers to apply sanitization and organization logic to data packets *as they are received*, paving the way for more traditional analyses of the collected data once the study is over.

The ARCHIE++ post-processing workflow is depicted in Figure 5. The data packets collected by the framework are shown on the left, where the contents of each packet are described by Table 2: the date and time the files were collected, the ID of the active configuration at the time of collection, the user's experience rating, the user's list of observed issues, the raw screenshot, the augmented screenshot, and the FPS trace for that interval. As these packets are collected, ARCHIE++ applies a set of "analyzer" scripts to sanitize, organize, or otherwise reason about their contents. These analyzer scripts take the form of Firebase Cloud Functions; ARCHIE++ comes with some basic functions (such as the ones described in Section 5.3's case study), but developers are free to supplement these scripts with whatever logic makes sense for their project. Sample outputs are shown on the right side of Figure 5; the developers could maintain a database of statistics calculated from the information gathered, sanitize and sort the camera frame pairs into buckets based on their contents or testers' submitted issue lists, or plot graphs of trends in user feedback. This helps developers focus their attention on what matters to them, whether it be triaging runtime issues or preparing for more in-depth analyses of collected data.

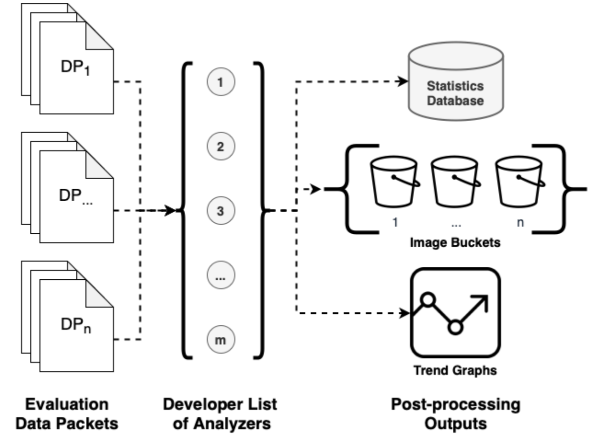


Fig. 5: Post-processing workflow for ARCHIE++ framework. A developer-selected list of analyzer functions are executed on data packets collected by the framework during the evaluation period, and used to generate an open-ended collection of outputs, such as databases of statistics, "buckets" of sanitized and sorted images, or graphs of trends in usability and issue data.

5 Case Studies

To demonstrate the usefulness of ARCHIE++, we present the following set of case study scenarios. Each scenario represents a problem focus identified in our motivating survey from Section 3 and highlights a different feature set within the ARCHIE++ framework: *comparison testing* using an avatar case study (Section 5.1), *runtime diagnostics* using an augmentation display and placement case study (Section 5.2), and *scalable long-term testing* using an interaction case study (Section 5.3). For simplicity, augmentation display and placement have been consolidated into a single scenario due to their similarity.

It should be noted that the functional segregation in the following scenarios are for illustrative purposes only, and *do not* represent the sum total of evaluations that developers can run using ARCHIE++. For example, we anticipate that a majority of developers will want to perform comparison testing, as it represented 60% of papers surveyed in Section 3; however, this does not preclude those developers from also conducting runtime diagnostics and/or scalability testing if they wish. The heart of the ARCHIE++ framework is developer configurability; developers are free to leverage any or all of these features in their own evaluations as it make sense for their project and problem focus.

5.1 Comparison Testing

Our first case study showcases the ability of the ARCHIE++ framework to assist developers with *comparison testing*, where testers are asked to review multiple interfaces and select their preferred option. In our motivating study, we observed that a significant portion (60%) of papers conducted some sort of comparison testing, which is a standard way of identifying tester preference and improving on existing methodologies.

Scenario Overview: Consider, therefore, a research team which seeks to understand whether virtual companions

```
{
  "config_ids": [
    "avatar_companion_cat",
    "avatar_companion_dog",
    "avatar_companion_human",
    "avatar_companion_voice_only"
  ],
  "config_test_period": {
    "value": 24,
    "denomination": "hours"
  },
  "collect_raw_frame": "false",
  "collect_aug_frame": "false"
}
```

Fig. 6: Sample manifest file for the Comparison Testing case study. Iterates between four unique interface configurations at 24hr intervals. For this case study, raw and augmented camera frames are not being collected.

```
@Override
public void OnConfigSelected(string configID){
  setAvatarForSelectedConfig(configID);
  setAudioForSelectedConfig(configID);
  respondToSwitch = true;
}

public void OnUpdate() {
  if (respondToSwitch) {
    displaySelectedAvatar();
    playSelectedAudio();
    respondToSwitch = false;
  }
}
```

Fig. 7: Developer-provided implementation for framework-required events (`OnConfigSelected()`) to satisfy the Comparison Testing case study, and display the selected resources when app next executes (`OnUpdate()`).

can reduce loneliness in older adults at home. Comparison testing is enabled within the ARCHIE++ framework using the system manifest file. Figure 6 shows a sample manifest for the avatar comparison scenario; it includes identifiers for the configurations they wish to test (cat, dog, human, voice-only), as well as the desired length of time to test each one (24 hours), and whether to collect raw and augmented frames from the camera feed (false, periodically collecting usability feedback will be sufficient). ARCHIE++ then takes over runtime configuration switching and feedback collection throughout the course of the evaluation period.

Framework Benefits: ARCHIE++ assists developers engaging in comparison testing by managing the rotation of multiple test conditions for each individual tester. This is accomplished by raising a dedicated event during runtime when it is time to switch to a new configuration (Figure 7, `OnConfigSelected()`), which developers can respond to in whatever way makes sense for their application. For this case study, the developers respond to the configuration change by preloading the audio and visual resources for the newly selected avatar, and engaging them when the application next executes (Figure 7, `OnUpdate()`). This event handler could just as easily be used to swap object recognition or label placement algorithms, label formatting and color palettes, and any other application-specific task. While traditional comparison techniques such as A/B testing require large bodies of testers where each tester experiences and provides feedback on only one condition, developers using ARCHIE++ can shepherd each tester through all implementation options that the team has to offer in a single evaluation period. All data packets collected by ARCHIE++ are grouped by name of the configuration under which that data was gathered, so that developers can review how system performance and user feedback changes between configuration options.

5.2 Runtime Diagnostics

For our second case study, we showcase the ability of the ARCHIE++ framework to facilitate *runtime diagnostics*, where data collected during the evaluation period is used to debug issues experienced therein. In our motivating study, we observed that the great majority (91%) papers used some sort of written usability survey or post-evaluation interview to gather feedback from their participants. While these approaches are good at giving general usability feedback, they cannot identify exactly how the system performed at runtime to generate that feedback.

Scenario Overview: Consider, therefore, a research team that has developed a cultural heritage application to label points of interest (POIs) in walking tours of a given city. They are ready to deploy their system for testing around town, and are particularly interested in whether the labels are conveniently placed and have sufficient contrast against testers' backgrounds. Figure 8 shows examples of how placement and contrast can affect user experience: poorly placed augmentations can cover up important real-world information, while augmentations with poor contrast may be unreadable against certain backgrounds. To test their system for these types of conditions, the researchers first configure their instance of ARCHIE++ to collect camera frames by setting the `collect_raw_frame` and `collect_aug_frame` properties in the system manifest file to "true". The framework then collects these images from the camera data stream automatically during runtime.

$$R = \sum_{i=1}^{\mu} \sum_{j=1}^{\nu} \alpha_1 O(i,j) \cdot S(i,j) + \alpha_2 O(i,j) \cdot E(i,j) \quad (1)$$

After the evaluation period is over, the research team can leverage ARCHIE++'s post-processing workflow to evaluate the pairs of camera frames for label placement and contrast, using the steps outlined in Figure 9. First, the raw and augmented images are compared, to identify and extract the augmentations from the scene (called the *overlay map*). Then the raw frame is processed to identify edges and areas of semantic importance within the scene, called the *edge map* and *saliency map*, respectively. Finally, the quality of augmentation placement (R) is calculated using Eq. 1, where $\mu \cdot \nu$ is the image size, $O(i,j)$ represents the overlay map indicating the region occupied by the projection of the augmented content on the image plane. $S(i,j)$ represents the saliency map, and $E(i,j)$ is the edge map. If R is *above* a given threshold, the augmentation placement is determined to be poor. Contrast can be determined using standard procedures, where the brightness of the augmentation pixels are compared with the brightness of the surrounding pixels.

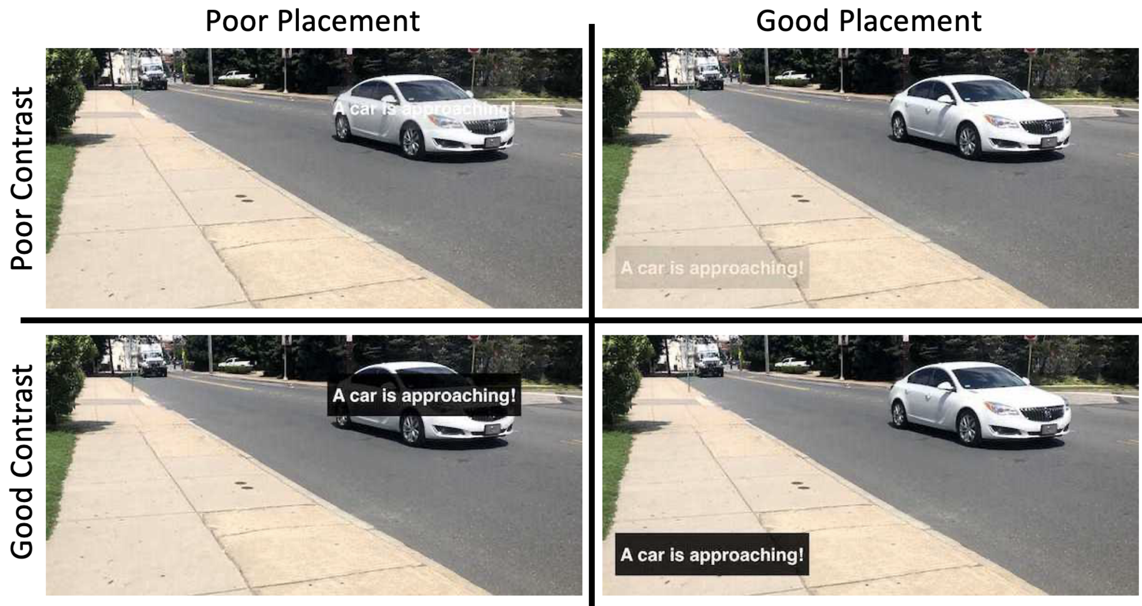


Fig. 8: Buckets with sample frames showing different combinations of augmentation placement and contrast quality.

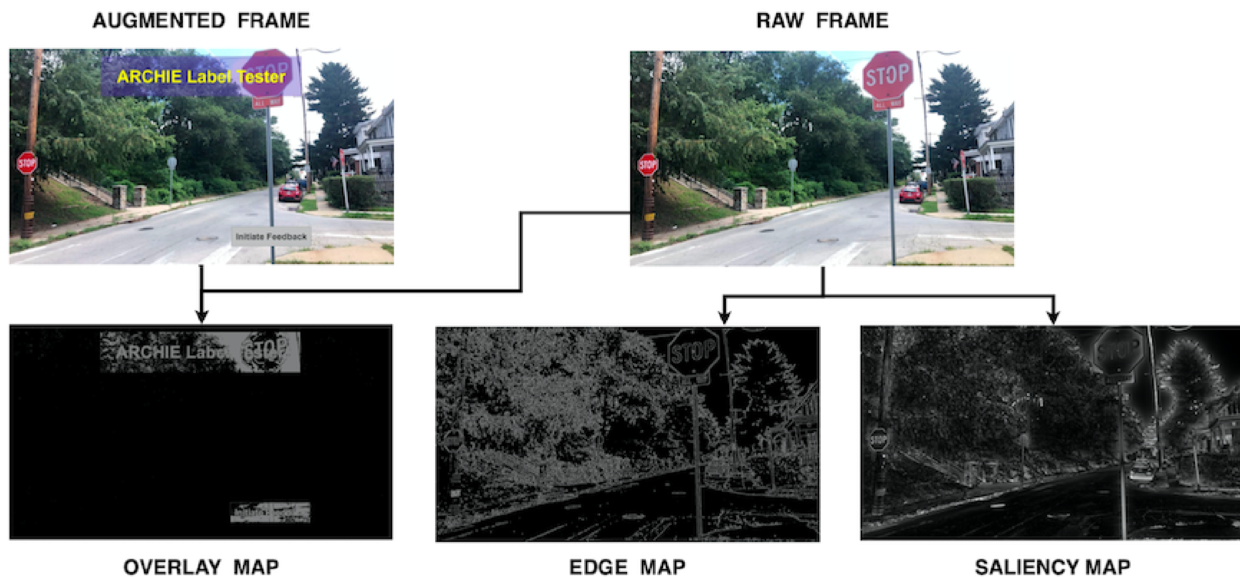


Fig. 9: Use of camera frame pairs to gauge system performance in the Runtime Diagnostics case study. We first compute the image difference between the two images (raw and augmented) to extract the overlay map (e.g. pixels that are occupied by the overlaid AR content), then we compute the edge and saliency maps from the raw image. All three maps are then fed into the layout quality estimator.

A contrast score *below* a given threshold is classified as poor. Labeled image pairs can then be assigned to "buckets" (such as those shown in Figure 8), and addressed by the research team in groups.

Framework Benefits: While standard usability surveys such as the System Usability Scale (SUS) [46] and the NASA Task Load Index (TLX) [11] can identify situations in which users experienced poor placement or readability of augmentations, they are unable to provide any systems-level assistance in diagnosing or debugging those issues. This problem is particularly prevalent in mobile AR systems running on smartphones or tablets, where runtime conditions such as ambient light levels, viewing angles, content and color of the image background, and number of active processes running on the device can all have direct impacts on system performance and the quality of augmentation display. By leveraging runtime files collected by ARCHIE++, developers can perform diagnostic tasks such as identifying performance dips using the FPS traces, or calculating hard metrics such as placement and contrast scores described above using the raw/augmented image pairs. Further, all of these data points can be cross-referenced with individual user feedback records for those exact instances. Traditional usability feedback surveys are unable to provide this kind

```
const functions = require('firebase-functions');
const admin = require('firebase-admin');
admin.initializeApp();

exports.aggregateFeedback = functions.storage.object().onFinalize(async (object) => {
    score = parseFeedbackScoreFromPacket(object.data);
    issues = parseIssueListFromPacket(object.data);
    environmentLabel = classifyEnvironmentFromRawImage(object.data);
    appendToRunningFile(getCurrentTimestamp(), score, issues, environmentLabel);
});

exports.generateGraphs = functions.pubsub.schedule('every day 00:00').onRun(async context => {
    generateScoreDataGraphFromRunningFile();
    generateIssueDataGraphFromRunningFile();
});
```

Fig. 10: Sample ARCHIE++ post-processing analyzer functions for Long-term Testing case study. "`aggregateFeedback()`" parses data (timestamp, usability score, list of issues encountered, environmental context label) from feedback packets as they are uploaded to Firebase, and writes it to a running log. "`generateGraphs()`" executes every night at midnight to plot graphs from the running data log, and write those graphs to image files.

of information.

5.3 Scalable Long-term Testing

Our third case study showcases the ability of the ARCHIE++ framework to support developers engaging in *scalable long-term testing*, in which evaluations span large groups of testers for long periods of time. In our motivating study, a significant portion of papers conducted user studies with fewer than 25 people (68%) with each tester spending only a single day on the task (79%). This is because recruiting, supporting, and debriefing user study participants is expensive and time-consuming for both the participants and the study administrators. Platforms such as Mechanical Turk can facilitate testing with larger groups of people, but typically only for short periods of time.

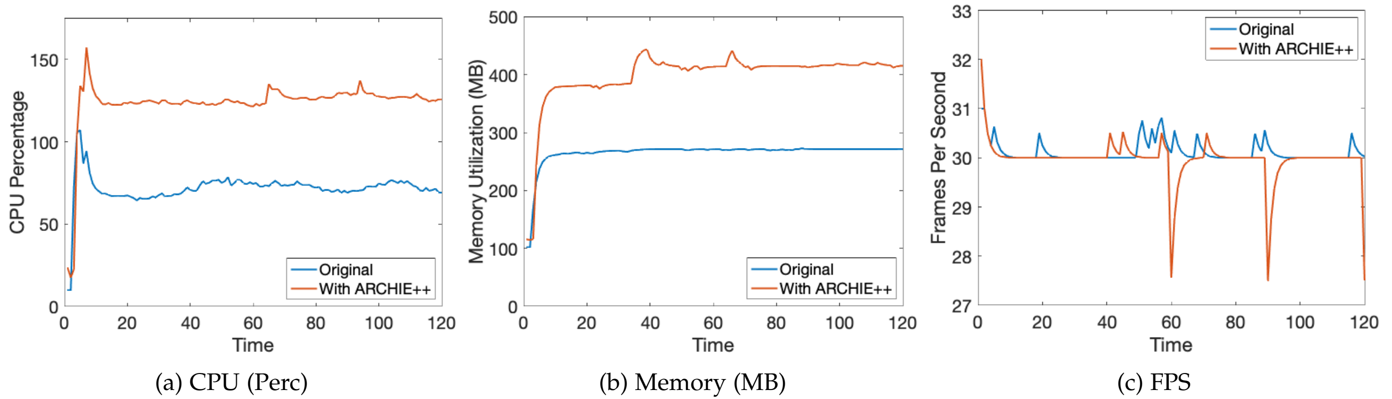
Scenario Overview: Consider, therefore, a research team that has developed a new method for using smartwatch IMU sensors to perform free-form gesture tracking as an input method for AR systems. They figure configure their instance of ARCHIE++ by disabling comparison testing (by leaving the `config_ids` and `config_test_period` fields blank within the system manifest file), and enabling frame collection (by setting the `collect_raw_frame` and `collect_aug_frame` fields to "true"). They then configure their ARCHIE++ post-processing environment with analyzer functions such as those shown in Figure 10, which take the form of Firebase Cloud Functions that operate on the data packets uploaded from the framework.

The first function (`aggregateFeedback()`) fires whenever a new data packet is uploaded to Firebase from the AUT. It unzips the packet, extracts the user feedback JSON file and raw image, and parses out the information that the team is interested in (specifically, the usability score, list of issues encountered, and environment classification label). These values, along with the current timestamp, are then appended to a running log. The second function (`generateGraphs()`) is configured to run every night at midnight, taking subsets of the data captured by the running log file and generating graphs charting such trends as usability scores over time or the most highly rated issues for a given month. These functions run autonomously for as long as the developers need them to; at any time, they can log in to the Firebase console and access these graphs to see the latest trends.

Framework Benefits: The benefit of long-term testing is that it gives a more realistic understanding of how testers interact with and feel about a system. This is because testers are able to experience the system over a variety of tasks, moods, and contexts, which can highlight usability concerns that were not apparent in initial short-term, constrained laboratory testing. However, relying only on traditional feedback methodologies such as journals or periodic surveys and interviews when conducting long-term testing can lead to a lower quality of feedback as testers are forced to recall system interactions instead of logging them in the moment. By combining *in situ* collection of performance and feedback data with cloud-based file storage, ARCHIE++ supports developers conducting long-term testing with large groups in supervising their participants and observing how usability feedback changes over time. Generating graphs from running log files, as described in the above case study, is just an example of what ARCHIE++ can offer with the help of Firebase Cloud Functions. Developers could write functions to perform any number of tasks, such as calculate a running metric and then email themselves when it falls above or below a certain threshold, or to classify and sanitize images as they come in to protect user privacy. While our prior work [33] performed all framework operations locally, this new distributed architecture allows for research and development teams to test remotely and at scale, with all collected artifacts accumulating in a single central location, agnostic of the number, type, or version of devices being used - a feature which is highly useful in the era of COVID19.

6 EVALUATIONS

In designing our evaluations, our goal was to demonstrate, not only that ARCHIE++ can operate within a range of AR system operations, but also that ARCHIE++ is able to perform its duties without negative impact to the application-under-test. It is crucial for a testing tool to remain transparent from the tester's perspective, and for any feedback that a tester provides to be a result of the application they are



(a) CPU (Perc) (b) Memory (MB) (c) FPS

Fig. 11: Resource consumption over time for **Environment** test condition before and after incorporating ARCHIE++

testing and not ARCHIE++. Additionally, we want to ensure that ARCHIE++ is manageable for developers to integrate into existing systems. For the two test conditions (described below), each code base required only 97 additional lines of code, whereas the prototypes evaluated in our prior work [33] required anywhere from 150 to 350 additional lines of code (a 35 - 72% decrease).

6.1 Test Set-up

One of our evaluation goals was to show how ARCHIE++ impacts system performance when performing different AR operations. To this end, we tested using applications configured to perform two types of sensing tasks:

- **Environment**: For the Environment condition, the system utilizes ground plane recognition from Vuforia. It presents a reticule on the ground plane, and places a 3D label reading "Hello ARCHIE" when the user touches the plane.
- **Object**: For the Object condition, the system utilizes 3D object recognition offered by Vuforia. (For our purposes, we used a coffee mug as shown in Figure 1.) The system then automatically detects the object when it comes into camera view, and places a 3D label reading "Hello ARCHIE" in front of the object.

These applications (Environment and Object) were considered our "original" conditions. Once the initial test conditions were established, we modified the code bases to incorporate ARCHIE++, yielding with two new applications. For these new conditions (With ARCHIE++), the labels were set to change font sizes every 10 seconds (from small to large), and the feedback form to display every 30 seconds. All four applications were implemented using the Unity3D IDE and executed on an iPhone 8 Plus with a six-core A11 processor, 3GB of memory, 64GB of storage, and a full HD screen (1920 x 1080 pixels). We elected to use a smartphone as our runtime device because it can also represent a head-mounted or augmented windshield display if a tester has the correct mount.

6.2 System Overhead

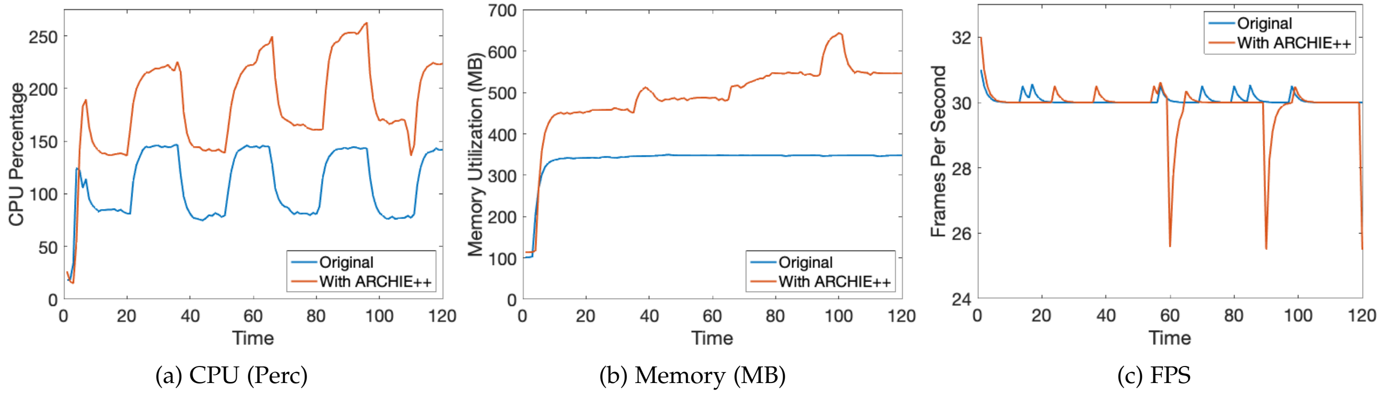
For our system overhead evaluations, we compared the runtime performance of the original Environment and Object test conditions with those counterparts that incorporated ARCHIE++. The test procedure involved 15 second intervals of pointing the device camera toward a recognized target (i.e. ground plane after touching or the pre-scanned mug), and a "neutral" view (i.e. a website with a stopwatch timer). These intervals were repeated for a 2 minute period, yielding four sessions for each view. When testing with an ARCHIE++ version, the feedback form was also displayed and submitted every 30 seconds after the first minute.

The results of those trials can be found in Figures 11 and 12 respectively. The raw signals were sampled at a rate of 1 Hz, and have been de-noised with a weighted moving average function. Our first observation is that, even though the same testing procedure was utilized in both conditions, the resource consumption for the Environment condition stays relatively flat while the traces for the Object condition exhibit distinct peaks and valleys. This is because general-purpose ground plane detection is a consistent process executed on every frame, where, once a predefined target has been identified, per-frame recognition attempts cease and the library switches to tracking the identified target.

While the system does experience an increase in CPU and memory consumption when incorporating ARCHIE++, the user experience does not seem to be affected, as evidenced by the lack of impact to the FPS. There *is* a sudden drop in FPS when the feedback form is displayed, as evidenced by the dips in Figures 11c and 12c, but the decrease lasts for only a single reading and does not significantly impact the average overall (30.12 to 29.96 for the Environment condition and 30.07 to 29.9 for the Object condition). Based on this, and the fact that ARCHIE++ is intended to be a testing platform and not utilized in everyday application use, we consider the increased resource consumption to be permissible.

6.3 Network and Storage Overhead

We also evaluated the bandwidth requirements to transmit and store framework artifacts in Firebase. This is important to help maintain a balance between storing large bodies of data with their helpfulness during post-processing. For this evaluation we considered transmissions for both the Environmental and Object test conditions, with both "simple" (a plain white wall) and "busy" (a densely patterned rug) backgrounds. It should be noted that, since the artifacts



(a) CPU (Perc) (b) Memory (MB) (c) FPS

Fig. 12: Resource consumption over time for **Object** test condition before and after incorporating ARCHIE++

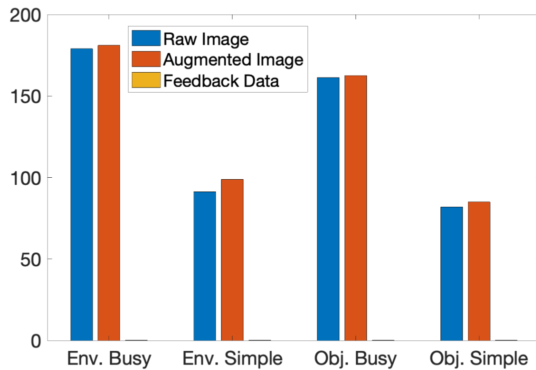


Fig. 13: Reported file sizes in kB when uploading to Firebase

are transmitted asynchronously from the test instance and evaluated *en masse* after the instance is over, we do not consider transmission latency in our evaluations.

Figure 13 shows the file sizes in KB when transmitting data packets to Firebase. All transmission packets were 360 KB or less, with corresponding contributions of the different files varying by condition and background. (It should be noted that the feedback file sizes are on the order of bytes rather than KB, and so are not visible in the graph.) While 360 KB is not an unreasonably large packet size, it can become burdensome to an AUT if many packets are being sent. Developers can control this to some extent by carefully managing the conditions under which feedback is being requested (described in Sections 4.3 and 4.4) and disabling the collection of either the raw or augmented input images if applicable. In future work, we wish to also explore the trade-offs of dynamically adjusting the resolution of the captured images during run-time, in order to balance transmission size and efficacy of the post-processing workflow.

7 Additional Discussion

Supplementing standard usability questionnaires. ARCHIE++ is intended to *supplement* rather than replicate or replace standard usability questionnaires such as the SUS [46] and the NASA TLX [11]. It does this by gathering data on system behavior, environmental inputs, and tester-observed issues during runtime. This additional data gives researchers a better understanding of contexts which precipitate poor usability feedback than usability questionnaires could on their own. The choice to prompt testers only for general usability scores and a list of observed issues was done to limit the impact of ARCHIE++ on the user experience. If the user was being stopped in the middle of their current task to answer lengthy questions on a mobile device, there could be significant impacts: at best, the tester's mental flow would be disrupted, while at worst, their situational awareness and overall safety could be affected. Therefore, we collect only what data we need during runtime, and rely on traditional methods to collect more generalized usability information.

Providing feedback with non-touchscreen modalities. While the current iteration of ARCHIE++ assumes the availability of a touchscreen interface for the purposes of this paper, the system can easily extend to display the feedback form on a companion device if the primary display is incompatible, such as an HMD or augmented windshield. In these cases, the feedback form could be displayed on a tethered smartphone or similar device to capture user input. In future work, we would like to explore the use of voice recognition to allow testers to speak their feedback rather than inputting it onto a physical form.

Supplementing standard data science toolkits. ARCHIE++ is also intended to *supplement* rather than replicate or replace traditional data science toolkits and workflows. Indeed, ARCHIE++'s post-processing workflow gives developers a "first line of defense" in terms of sanitizing and organizing data as it comes in, smoothing the way for more in-depth analyses later. Developers could apply, for example, a filter based on the algorithms presented in [42], where images taken in sensitive areas such as bathrooms and bedrooms are automatically dropped from storage, or those presented in [49], where individual sensitive items in pictures such as faces or license plates are proactively blurred out. Developers could, alternatively, use the data packets coming in to extract or calculate additional data points that construct a separate supplementary data set, such as the edge and saliency maps presented in Section 5.2. All of this data could then be processed using traditional analysis methods as necessary.

Privacy concerns when collecting camera frames. While there are legitimate privacy concerns inherent in collecting images from users' devices, ARCHIE++ is ultimately a test-

ing tool; as such, we can make certain assumptions. First, we assume that developers have obtained the appropriate IRB approval, as well as permission from testers to collect images as part of their study. Second, we assume that users will be interacting with the system only for a pre-determined amount of time, limiting the amount of data captured by the system. Finally, we assume that research teams utilizing ARCHIE++ are taking the appropriate privacy precautions, such as (but not limited to), screening or sanitizing images as necessary using techniques such as those presented in [42], [49].

Moving beyond AR. Finally, we would like to address the inevitable question of whether ARCHIE++ can support VR applications. A team wishing to utilize ARCHIE++ to evaluate a VR application could absolutely do so; the client-side Unity plugin and device-agnostic Firebase backend could support it. The team could utilize ARCHIE++ to compare functional options such as different types of avatars, different methods of data visualization, and more. However, the current input method for the feedback form would need to be updated, as it assumes a touch-screen interface, either through the device itself or a companion device, visible through a see-through HMD. We also would anticipate that some of the data collected by ARCHIE++ would be somewhat redundant to developers, as the testers are constrained to a predetermined virtual environment, rather than moving through an open-ended physical one.

8 Conclusions

In this paper, we presented ARCHIE++, the Augmented Reality Computer-Human Interaction Evaluator framework for conducting AR system testing and tester feedback collection in the wild. We demonstrated the need for scalable, reproducible usability testing in AR, and described our system architecture to support this need. We also demonstrated how ARCHIE++ can be incorporated into an existing AR system with acceptable system overhead, and presented a real-world case study for our framework. In the future, we would like to investigate real-world trade-offs between screenshot size and the accuracy of post-processing analysis, and explore security implications of keeping in-the-wild camera frames in third-party cloud storage.

Acknowledgments

Dr. Ling was partially supported by NSF grants 2006665, 2128350 and 2128187.

References

[1] Compositor mirror — oculus documentation. https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-compositor-mirror/. Accessed: 2020-09-09.
[2] Oculus debug tool — oculus documentation. https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-debug-tool/. Accessed: 2020-09-09.
[3] Perception simulation - mixed reality — microsoft docs. https://docs.microsoft.com/en-us/windows/mixed-reality/perception-simulation. Accessed: 2019-10-22.
[4] Performance heads-up display — oculus documentation. https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-hud/. Accessed: 2020-09-09.
[5] Profiler tool reference — unreal engine documentation. https://docs.unrealengine.com/en-US/Engine/Performance/Profiler/index.html. Accessed: 2020-09-09.
[6] Reality composer - augmented reality - apple developer. https://developer.apple.com/augmented-reality/reality-composer/. Accessed: 2020-09-09.
[7] Testing your app on hololens - mixed reality — microsoft docs. https://docs.microsoft.com/en-us/windows/mixed-reality/testing-your-app-on-hololens. Accessed: 2020-09-09.
[8] Using the hololens emulator - mixed reality — microsoft docs. https://docs.microsoft.com/en-us/windows/mixed-reality/using-the-hololens-emulator. Accessed: 2019-10-22.
[9] Using the windows mixed reality simulator - mixed reality — microsoft docs. https://docs.microsoft.com/en-us/windows/mixed-reality/using-the-windows-mixed-reality-simulator. Accessed: 2020-09-09.
[10] VR performance optimization guide — oculus documentation. https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-performance-opt-guide/. Accessed: 2020-09-09.
[11] N. Aeronautics and S. Administration. TLX @ NASA ames - home. https://humansystems.arc.nasa.gov/ groups/TLX/. Accessed: 2020-07-23.
[12] M. Akçayır and G. Akçayır. Advantages and challenges associated with augmented reality for education: A systematic review of the literature. *Educational Research Review*, 20:1–11, 2017.
[13] J. An, L.-P. Poly, and T. A. Holme. Usability testing and the development of an augmented reality application for laboratory learning. *Journal of Chemical Education*, 97(1):97–105, 2019.
[14] S. Barbieri, G. Vettore, V. Pietrantonio, R. Snenghi, A. Tredese, M. Bergamini, S. Previato, A. Stefanati, R. M. Gaudio, and P. Feltracco. Pedestrian inattention blindness while playing pokémon go as an emerging health-risk behavior: a case report. *Journal of medical internet research*, 19(4):e86, 2017.
[15] S. Bernhardt, S. A. Nicolau, L. Soler, and C. Doignon. The status of augmented reality in laparoscopic surgery as of 2016. *Medical image analysis*, 37:66–90, 2017.
[16] M. Billinghurst, A. Clark, and G. Lee. A survey of augmented reality. *Foundations and Trends in Human-Computer Interaction*, 2015.
[17] F. Bonetti, G. Warnaby, and L. Quinn. Augmented reality and virtual reality in physical and online retailing: A review, synthesis and research agenda. In *Augmented reality and virtual reality*, pp. 119–132. Springer, 2018.
[18] M. Brehmer, B. Lee, P. Isenberg, and E. K. Choe. A comparative evaluation of animation and small multiples for trend visualization on mobile phones. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):364–374, 2019.
[19] P. W. Butcher, N. W. John, and P. D. Ritsos. VRIA: A web-based framework for creating immersive analytics experiences. *IEEE Transactions on visualization and computer graphics*, 27(7):3213–3225, 2020.
[20] P. Chen, X. Liu, W. Cheng, and R. Huang. A review of using augmented reality in education from 2011 to 2016. In *Innovations in smart learning*, pp. 13–18. Springer, 2017.
[21] L. Costa, M. Aliannejadi, and F. Crestani. A tool for conducting user studies on mobile devices. In *Proceedings of the 2020 Conference on Human Information Interaction and Retrieval*, pp. 462–466, 2020.
[22] S. G. Dacko. Enabling smart retail settings via mobile augmented reality shopping apps. *Technological Forecasting and Social Change*, 124:243–256, 2017.
[23] A. Ewais and O. D. Troyer. A usability and acceptance evaluation of the use of augmented reality for learning atoms and molecules reaction by primary school female students in palestine. *Journal of Educational Computing Research*, 57(7):1643–1670, 2019.
[24] P. Fraga-Lamas, T. M. Fernández-Caramés, Ó. Blanco-Novoa, and M. A. Vilar-Montesinos. A review on industrial augmented reality systems for the industry 4.0 shipyard. *Ieee Access*, 6:13358–13375, 2018.
[25] Google. Firebase. https://firebase.google.com. Accessed: 2021-09-26.
[26] C. L. Hughes, C. Fidopiastis, K. M. Stanney, P. S. Bailey, and E. Ruiz. The psychometrics of cybersickness in augmented reality. *Frontiers in Virtual Reality*, 1:34, 2020.
[27] S. Inc. Snapchat - apps on google play. https://play.google.com/store/apps/details?id=com.snapchat.android. Accessed: 2020-08-23.

[28] M. Y. Ivory and M. A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys (CSUR)*, 33(4):470–516, 2001.

[29] B. Joseph and D. G. Armstrong. Potential perils of peri-pokémon perambulation: the dark reality of augmented reality? *Oxford medical case reports*, 2016(10), 2016.

[30] R. M. Kelly, H. S. Ferdous, N. Wouters, and F. Vetere. Can mobile augmented reality stimulate a honeypot effect? observations from santa's lil helper. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2019.

[31] S. Kim, M. A. Nussbaum, and J. L. Gabbard. Influences of augmented reality head-worn display type and user interface design on performance and usability in simulated warehouse order picking. *Applied ergonomics*, 74:186–193, 2019.

[32] E. M. Kolasinski. *Simulator sickness in virtual environments*, vol. 1027. US Army Research Institute for the Behavioral and Social Sciences, 1995.

[33] S. M. Lehman, H. Ling, and C. C. Tan. ARCHIE: A user-focused framework for testing augmented reality applications in the wild. In *2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 903–912. IEEE, 2020.

[34] D. Lindlbauer, A. M. Feit, and O. Hilliges. Context-aware online adaptation of mixed reality interfaces. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pp. 147–160, 2019.

[35] A. Mottelson and K. Hornbæk. Virtual reality studies outside the laboratory. In *Proceedings of the 23rd acm symposium on virtual reality software and technology*, pp. 1–10, 2017.

[36] M. Nebeling, M. Speicher, X. Wang, S. Rajaram, B. D. Hall, Z. Xie, A. R. Raistrick, M. Aebersold, E. G. Happ, J. Wang, et al. MRAT: The mixed reality analytics toolkit. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2020.

[37] R. Palmarini, J. A. Erkoyuncu, R. Roy, and H. Torabmostaedi. A systematic review of augmented reality applications in maintenance. *Robotics and Computer-Integrated Manufacturing*, 49:215–228, 2018.

[38] L. Poretski, O. Arazy, J. Lanir, S. Shahar, and O. Nov. Virtual objects in the physical world: Relatedness and psychological ownership in augmented reality. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2019.

[39] I. Radu, B. MacIntyre, and S. Lourenco. Comparing children's crosshair and finger interactions in handheld augmented reality: Relationships between usability and child development. In *Proceedings of the The 15th International Conference on Interaction Design and Children*, pp. 288–298, 2016.

[40] P. Roberto, F. Emanuele, Z. Primo, M. Adriano, L. Jelena, and P. Marina. Design, large-scale usage testing, and important metrics for augmented reality gaming applications. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 15(2):1–18, 2019.

[41] L. R. Rochlen, R. Levine, and A. R. Tait. First person point of view augmented reality for central line insertion training: A usability and feasibility study. *Simulation in healthcare: journal of the Society for Simulation in Healthcare*, 12(1):57, 2017.

[42] R. Templeman, M. Korayem, D. J. Crandall, and A. Kapadia. PlaceAvoider: Steering first-person cameras away from sensitive spaces. In *NDSS*, pp. 23–26. Citeseer, 2014.

[43] Unity3D. Platform development. https://docs.unity3d.com/Manual/PlatformSpecific.html. Accessed: 2021-09-04.

[44] Unity3D. Unity3D game engine. www.unity3d.com. Accessed: 2021-09-04.

[45] Unity3D. XR suggested platforms. https://docs.unity3d.com/Manual/XR.html. Accessed: 2021-09-04.

[46] Usability.gov. System usability scale (SUS). https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html. Accessed: 2020-07-23.

[47] P. Vávra, J. Roman, P. Zonča, P. Ihnát, M. Němec, J. Kumar, N. Habib, and A. El-Gendi. Recent development of augmented reality in surgery: a review. *Journal of healthcare engineering*, 2017, 2017.

[48] A. Vovk, F. Wild, W. Guest, and T. Kuula. Simulator sickness in augmented reality training using the microsoft hololens. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pp. 1–9, 2018.

[49] E. Zarepour, M. Hosseini, S. S. Kanhere, and A. Sowmya. A context-based privacy preserving framework for wearable visual lifeloggers. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pp. 1–4. IEEE, 2016.

Sarah M. Lehman (student member, IEEE) received her B.A. in Computer Science from Messiah College in 2010 and her Masters in Software Engineering from the Pennsylvania State University in 2015. She is currently completing her final year as a PhD student of Computer Science at Temple University where her research focuses on testing and usability problems of mobile augmented reality systems.

Semir Elezovikj received the B.S. degree in Computer Science from Jacobs University, Bremen, Germany in 2008. He received the M.S. degree in Computer Science from Temple University in 2014. He is currently a PhD Candidate in Computer Vision with the Department of Computer and Information Sciences at Temple. His research interests include computer vision, augmented reality, machine learning and multipath trajectory prediction in the domain of self-driving vehicles.

Dr. Haibin Ling received the B.S. and M.S. degrees from Peking University in 1997 and 2000, respectively, and the Ph.D. degree from the University of Maryland, College Park, in 2006. From 2000 to 2001, he was an assistant researcher at Microsoft Research Asia. From 2006 to 2007, he worked as a postdoctoral scientist at the University of California Los Angeles. In 2007, he joined Siemens Corporate Research as a research scientist; then, from 2008 to 2019, he worked as an Assistant Professor and then Associate Professor at Temple University. In fall 2019, he joined Stony Brook University as a SUNY Empire Innovation Professor in the Department of Computer Science. His research interests include computer vision, augmented reality, medical image analysis, machine learning, and human computer interaction. He received Best Student Paper Award at ACM UIST (2003), Best Journal Paper Award at IEEE VR (2021), NSF CAREER Award (2014), Yahoo Faculty Research and Engagement Award (2019), and Amazon Machine Learning Research Award (2019). He serves or served as Associate Editors for IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI), Pattern Recognition (PR), and Computer Vision and Image Understanding (CVIU). He has served as Area Chair various times for CVPR and ECCV.

Dr. Chiu C. Tan (member, IEEE) received the B.A. and B.S. degrees from the University of Texas at Austin in 2004, and the Ph.D. degree from the College of William and Mary in 2010. He is currently an Associate Professor with the Department of Computer and Information Sciences at Temple University. His research interests are in the areas of cyber security, mobile AR/VR, camera networks, smart health systems, and wireless network security (main 802.11, RFID, and sensor networks).