https://doi.org/10.2352/EI.2024.36.1.VDA-362
© 2024, Society for Imaging Science and Technology

RAIV: Researchable Archives for Interactive Visualizations

Hunter Price[1], John Duggan[1], Robert Sisneros[2], Tanner Hobson[1], James Hammer[1], James Osborne[1], Jian Huang[1]
[1]University of Tennessee, Knoxville, TN
[2]National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana Champagne, IL

Abstract

Web visualization dashboards are popular. We propose a system called RAIV that can capture and archive web visualizations into self-contained objects. RAIV also uses a client-server architecture to host and manage archived objects as online galleries, which users can use a standard web browser to experience without needing to install any additional software. RAIV supports intelligent search as well. When a search target has been found, RAIV can show the interaction path required to reach that target. We demonstrate RAIV's capability using a genomics web visualization system called KnowEnG from NCSA and publicly available census data visualizations from US Census.

1 Introduction

When data drives decisions, interactive visualization is important because it enables users to make discoveries and share insights to peers, stakeholders, decision makers, and the public alike.

After a user loads a dataset into an interactive visualization application, the app's functionality of analyzing that dataset is represented fully by the user interface (UI) of the app. Fundamentally, the user's interactive experience revolves around a cycle of triggering a UI functionality, seeing the visual display, and triggering another functionality to continue the cycle.

Furthermore, tasks done using visualization applications are often repetitive. Given the same sequence of interactions, the same visualization is produced. To this end, interactive functionalities to analyze a given dataset using a specific visualization app can be archived into an independent object and later recreated, as long as the collective settings of a visualization app's UI can represent the state of the visualizations. When this condition is met, the same visualization results of the same dataset are reproducible by using the same settings in the same application.

In a prior work [15], we have confirmed the possibility of capturing interactive visualization into archived objects, especially that such archived objects can be meaningfully reused in practical settings. Our method leveraged the mapping between the collective status of the visualization's UI widgets and its visual output (i.e. framebuffer). Using Tableau, ParaView and journalistic visualizations from NYTimes.com as examples, our method was able to capture a subset of the visualization app's interactive functionalities when analyzing a given dataset into a standalone object. The subset of the original app's functionalities to capture is customizable and user specified. That prior work is open source and called Loom. Technical details in Section 2.2.

While Loom's capturing process uses automated UI-bots, in a general desktop setting, the process to set up a Loom capture is labor intensive and error prone, chiefly because UI-bots require information about widgets, such as position, size, and kinds of events the widget responds to. Even though a human user who is creating the archived visualization can provide that information, the manual specification process is very costly. In this work, we propose to focus on web-based visualizations due to their widespread presence and also the fact that such widget-specific metadata can be harvested, and inferred, from each web application directly.

To archive web applications effectively, we have developed a new system called RAIV, which appears as a browser plugin and injects code into the Document Object Model (DOM) of a target web visualization. RAIV's injected code recognizes the UI widgets involved automatically, so that a user creating an archived visualization can provide the specifications easily. The result is that the specification process that previously took minutes can now be reduced to tens of seconds. RAIV's injected code also transparently sets up an automated UI-bot to orchestrate the capture process, which further alleviates the need for human participation.

Unlike Loom using a client-side-only architecture, the RAIV system employs a client-server architecture. While RAIV preserves Loom's ability to stay application agnostic. the client-server architecture and also because RAIV's injected code transparently instruments a UI-bot into the target web visualization, RAIV can greatly accelerate the speed at which the UI-bot can function. Of course, as the speed of archival increases, the number of archivable interactions increases too. As the size of Loom objects has increased, we have found it beneficial to also develop a shared repository of archived visualizations.

In addition, in a client-server architecture, we can easily use the RAIV server as a host of an online "gallery" of captured RAIV objects. In this regard, because a user browsing through the online gallery gets to experience the captured objects in a way that is intuitively very similar to using a YouTube and not requiring users to install any specific software. This capability simplifies how scientists can share interactive visualizations with peers. Like YouTube, finding objects of interest relies on a powerful search engine. Now that we harvest metadata from the application itself, we can take advantage of the additional metadata to create our own search engine for archived visualizations.

Lastly, because each RAIV object represents subsets of interactive functionalities of using a visualization app to analyze a specific dataset, it is easy to imagine a library of RAIV objects, for example, using KnowEnG [1], a genomics visualization application, to analyze different genomics datasets. We propose to further extend a repository of RAIV objects to also include abilities of both textual- and image-based search. After a search target has been found, the corresponding RAIV object can also reveal the

[1] `https://knoweng.org/`

interaction paths that a user can take to get to that search result, as if that user is just using that original KnowEnG application. This reproducibility capability has general implications.

RAIV enables two types of power users primarily: (i) developers of web visualizations can create hosted snapshots of user experiences, as a lower-cost alternative to provide user with access to limited analytical functionalities or even as a method for regression testing, and (ii) data analysts to quickly capture and record an analysis to keep for records as well as subsequent sharing with peers.

In summary, the contributions of this work are: (1) greatly improving the usability and reliability of the capturing method of Loom by focusing on web-based visualizations organized around the standard DOM structure, (2) extending Loom's client-side only architecture to be client-and-server and, with this extension, created a server that can efficiently encode Loom objects and host captured objects as a scalable online gallery, (3) based on the availability of a preliminary archival gallery, creating an initial method for scientific users to search into the gallery by performing text- or image-based queries and returning not only the search target, but also the interaction paths that can take a user to that search result as if the user were using the original visualization app.

The rest of the paper is organized in the following way. We describe related work in Section 2. Our system design is in Section 3. We discuss results and conclude in Sections 4 and Section 5, respectively.

2 Related Work

2.1 The Science Needs

Among science communities, archival and reproducible scientific results have been a frequent and national topic in high-profile publications in Science [12, 4, 11], in Nature [17, 14], and in NSF reports and Dear Colleague Letters [1, 2, 7, 3, 10].

The growing trend of convergence research brings out another challenge. According to the article "Before Reproducibility Must Come Preproducibility" [18] in Nature 2018 Special Issue on Challenges in Irreproducible Research: "Science may be described as the art of systematic oversimplification — the art of discerning what we may with advantage omit. ... Communicating a scientific result requires enumerating, recording and reporting those things that cannot with advantage be omitted." In other words, having the right context is crucial to understanding and evaluating scientific results. Thus, archiving each visual insight together with the proper context is necessary.

Because archiving visualizations in a form where a user could still interact with the visualization would be of great benefit, executable paper has been a decade-long vision shared by the visualization community together with many other scientific domains. This research area started with the foundational system named VisTrails [5, 6]. A plethora of research was published since, including the very recent paper in Nature's Communications Physics [8] that emphasized the need for the technology and infrastructure to support the idea of executable paper.

Most of these methods rely upon the possibility to preserve and recreate an execution environment using traditional platforms or even the cloud [16]. From a practical point of view, the cost, expertise required, and the need for clear and actionable standards are still barriers of widespread deployment. However, there are deeper barriers too, in particular as pointed out in the 2020 article "No Raw Data, No Science: another possible source of the reproducibility crisis" [13], which illustrates the urgency for and yet also the dearth and the difficulty of widely sharing the original data. Furthermore, as a basis of research, scientific datasets are the output of previous scientific processes and also the input of new scientific processes. When sharing data, better ways that allow would-be users to make discernment on their own is invaluable, because "science advances faster when people waste less time pursuing false leads" (opening statement, Nature - 2018 Special Issue on Challenges in Irreproducible Research).

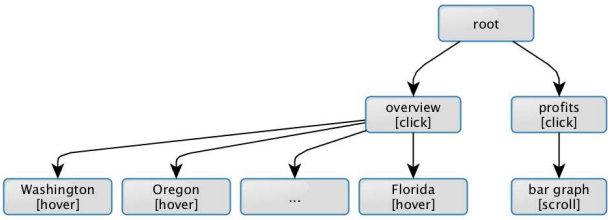

Figure 1: A simplified action tree example.

We propose to create archives of data visualizations as repositories of Loom objects, and hope to argue that the dataless manner in which Loom captures interaction can address the barriers of sharing the data. In addition, requiring only a standard web browser for a user to use and discern can address the barriers due to difficulties of faithfully recreating the computing environment used by the original research. In this work, we have developed an automated tool set to capture web based visualization dashboards at scale, a server architecture that can encode and host large numbers of Loom objects, and demonstrate that the result repositories can be used for both text- and image-based queries.

2.2 Loom Background

The premise of Loom is to consider visualization dashboards and applications as black boxes, and to focus solely on the relationship between the input (i.e. interactions / user actions) and the output (i.e. frame buffer). Obviously, the interaction space can be exponentially large, as is the potential space of output. Loom's perspective is that when a user interacts with complex software, only a small set of controls on the graphical user interface (GUI) are used at any time, which leads to each user session covering a rather narrow space of potential user action. Herein, we refer to this narrowed space as the interaction space, once defined, the corresponding output, i.e. the framebuffer space is accordingly defined.

Loom [15] showed that the interaction space can be captured together with the corresponding output space through automation, and that the mapping between the two spaces can be stored with great space efficiency and reused with great reliability. Loom was not ready for widespread adoption and use, because the process to create a Loom object is manual, time consuming, and error prone.

Just as with general user interfaces, interactions with a visualization are hierarchical in nature. With every action, a user can get a new set of options to interact with the application. Inspired by behavior trees [9], in Loom we model action hierarchies of an

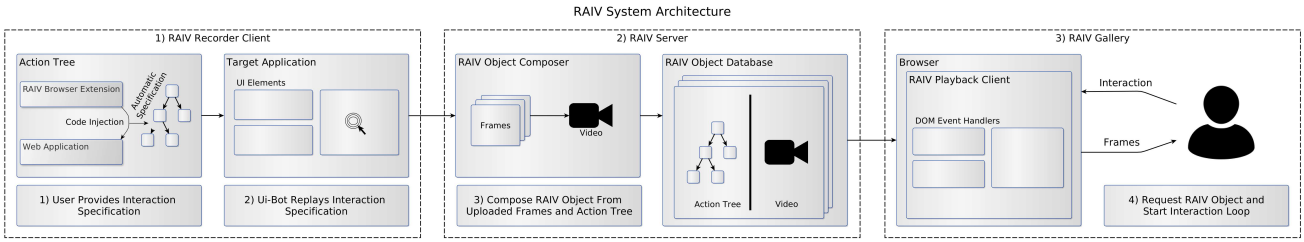

Figure 2: Overview of the RAIV system architecture. (1) The RAIV Recorder Client is injected into the Web Application to harvest the DOM. A user will specify the actions they would like to capture. Once ready, a user will start the capture sequence. A UI-bot will replay the specified actions and take screenshots after each action. These frames and the action specifications will be sent to the RAIV Server. (2) The RAIV Server will receive the frames from the RAIV Recorder Client and compose the RAIV object. The RAIV object is stored along with other recorded archives to be displayed in the RAIV Gallery. (3) A user will navigate to the RAIV Gallery and request to view a specific RAIV object. The RAIV Gallery will then set up the necessary event handlers and begin the interaction loop.

interactive visualization as an *action tree* (Figure 1). Loom then traverses the tree automatically, and uses a UI-bot to takes those actions and capture the visual responses associated with the actions. The pairs of action (i.e. input controls) and response (i.e. visualizations) are then stored together. For space efficiency, and also more critically, for seamless use and sustainable deployment, we use modern video technology as the container to package the action-response (i.e. interaction-framebuffer) data. Our examples in [15] chose MP4 with H.264 as the container, but WebM and H.265 should work as well.

When using a Loom object, Loom's browser-based viewer reconstructs the visualization application using HTML's DOM event handlers. Based on the user's interactions in the browser, appropriate visual responses (i.e. framebuffer from the Loom object) are selected by Loom and displayed. To end-users, this experience is as if they are truly interacting with the original visualization and data. In this way, the availability and accessibility of the recorded user experience do not depend on the original data or software.

3 The RAIV System

RAIV is designed as an extension and improvement on the Loom [15] architecture with the goal of improving ease of use and enabling user to navigate 10s to 100s of archived visualizations.

3.1 Design Objectives

To illustrate the design objectives of RAIV, it is important to understand its foundation. Loom [15] defines a data model for archived visualization that partitions a visualization application into a set of interactions and a set of visual outputs.

To capture an archived visualization, Loom employs a UI-bot which injects synthetic user inputs into an application, waits some time, and then captures the visual output of the application using standard operating system screenshot mechanisms.

A core challenge behind Loom's approach is that it lacks automation to help users specify the capturing process. It treats an application as a black box and assumes nothing about the underlying software. This inherent lack of information constrains the recorder, limiting the variety of interactions it can automate. These shortcomings include but are not limited to: requiring the user to manually specify bounding boxes of widgets, waiting a set amount of time before a screenshot is taken increasing the total capture time of a recording, no way of automatically detecting the type of a widget, etc.

There is no centralized and standardized way to manage Loom archives. If a user wanted to playback someone's Loom object, there is no hosted solution, making it inconvenient. This inconvenience hinders wide spread adoption, as without a platform to share and store recordings, users put more work on themselves to ensure the archive reaches its intended audience.

Finally, even if you had a large collection of Loom objects, it is impossible to enumerate every frame and interaction. You must selectively filter and search the frames. To this point, there has been no system introduced to intelligently navigate through these visual insights.

Goals. This work has two main goals for the RAIV Architecture. First, (**G1**) the system should alleviate additional manual labor in the recording process, to help save developers more time and effort when capturing applications. Second, (**G2**) the system should enable users to navigate through and query our database of visual analytics.

Objectives. With these goals in mind, we propose the following specific system objectives.

(O1) Provide robust and efficient recorder automation. The RAIV recorder must introduce automation to limit the amount of manual labor a user must undertake for common tasks to support (*G1*). (**O1.1**) Automation should reduce the effort required to specify the action-tree of a recording. (**O1.2**) It should speed up the capture phase by harvesting the DOM to automatically recognize when the web application has updated. (**O1.3**) The recorder should automatically upload a recording to the RAIV Gallery.

(O2) Enable scalability together with accessibility. The RAIV Server and Gallery must provide a centralized location that holds and serves Loom Objects. By having a central data repository we can begin to support navigation and queries against this database (*G2*). (**O2.1**) The gallery must allow a user to view and play the collection of archives. (**O2.2**) The RAIV Server should also allow a user to easily share Loom recordings.

(O3) Support intelligent search capabilities. The RAIV Gallery must provide an interface that allows a user to intelligently navigate and search through a collection of Loom objects (*G2*). (**O3.1**) The system must first be able to index into Loom objects via a text based query using tags contained with in the DOM when capturing an application. (**O3.2**) The system must also al-

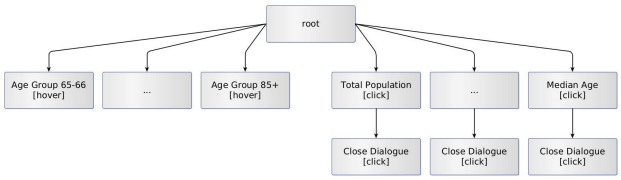

Figure 3: An action tree for the United States Census Bureau example. The RAIV Recorder builds an action map such as this when the user specifies which actions to record.

low users to find similar Loom objects and frames within them using an image-based query on the Loom frames themselves.

3.2 System Architecture

Figure 2 shows an overview of the RAIV system. RAIV has two core components: (i) the RAIV client-side recorder which is packaged as a Chrome extension, and (ii) the RAIV gallery server, which hosts and manages the captured Loom objects.

The RAIV client-side recorder (Figure 2.1) serves two purposes. First, it enables a user to specify an action tree that encompasses a subset of the interactions within a web application. Second, it replays the action tree specification with a UI-bot, and, for each action, it captures the frame buffer and streams it directly to the RAIV gallery server.

The RAIV gallery server (Figure 2.2 and 2.3) is a centralized service designed to collect and serve archives captured by the RAIV recorder. The gallery provides several core functionalities. First, it serves as a centralized database of Loom objects captured by the RAIV recorder and streamed directly to the server. Second, it provides a Youtube like gallery that enables users to easily navigate through and share user created archives. Next, it allows users to open and use Loom recordings with a single click on the gallery. Finally, it enables users to intelligently search through our database of visual insights.

3.3 The RAIV Client-Side Recorder

We design the RAIV recorder to fulfill the objective of provid robust and efficient recorder automation. RAIV implements the recorder as described in Loom. That is, a significant portion of actions taken within interactive visualizations can be modeled as a hierarchical directed acyclic graph. When an action is taken on an application, additional options for actions may then be revealed. Loom models this hierarchy of actions using an *action tree*; in this work we maintain the same abstraction.

In the following sections we provide a brief review of this abstraction along with our additions to Loom. We then describe the process a user will undergo to create a recording. Finally, we detail how the RAIV recorder harvests the DOM in the recording process.

3.3.1 Representing An Interactive Visualization

A RAIV object comprises a set of frames stored within an MP4 file and the action tree specification within a JSON file. On a high level, the action tree is a lookup to which frame will be displayed to a user. Each node in this hierarchy of interactions represents an index into a collection of frames gathered by the recorder. A node

Figure 4: The RAIV Recorder user interface. Users will see this UI when they start the Chrome extension. This interface allows a user to toggle whether or not they are recording, clear the recording, change server upload settings, set dependencies in the action specification, and begin the capture.

is also synonymous with an action. Each action within the action tree represents a widget with which a user interacts. Within this action object, we hold the following set of fields:

- **ActionType (String)** represents the type of action that a user can take on the object. Valid types include “click,” “hover,” “toggle,” and “slider.”
- **BoundingBox (List)** represents the rectangular area that a user can interact with this action.
- **Children (Object)** represents a list of elements dependent on this action. An example of this dependence are buttons that create dialogues or popup. Actions taken on resulting dialogue are only accessible when at their parent’s action.
- **Frame Position (Integer)** represents the index of the corresponding frame within the MP4. When an action is taken the playback will seek to this index.
- **Tags (String)** holds all of the text within the page. We use this further down the pipeline to semantically search through the frames.

This setup allows us to model complex sequences of interactions that may occur within an application. Figure 3 shows an example of an action tree for capturing a United States Census Bureau COVID-19 Impact Planning Report. The left side of the tree shows a chain of hover interactions on a bar chart within the interactive visualization. In this recording, these hover actions do not have any dependencies as they are connected directly to the root. The right side of the tree shows simple dependencies of buttons that create popup dialogues. The click actions spawn their dialogues; they each have their own children’s actions that are only accessible when a user has navigated to their parent.

In addition to the basic dependencies introduced in Loom, we introduce a concept called *independent actions*. Independent actions are those actions that should be accessible regardless of the current application state. A relevant example of independent actions is tooltips or hover events. Actions such as these, however small, significantly improve the dynamic feel of an interactive visualization.

The recorder supports such actions by allowing a user to specify a set of actions as independent. On capture time, each

action specified as independent is appended to all other non-independent actions.

3.3.2 Specifying A Recording

The general workflow of creating a recording is as follows. First, the user will navigate to the desired webpage and start the recorder (shown in Figure 4). The user will then specify which widgets to record, the action types, and dependencies. Then, the user will enter information about the server to upload to and click capture. A UI-bot will then iterate through the action tree and perform them on the webpage. The bot will take a screenshot of the webpage on each action and upload that image to the RAIV Gallery. Once the bot has iterated through the action tree, the Gallery will encode the images into an MP4 file and store it with the specified action tree.

3.3.3 Video Encoding

We package the screen captures of an application into a single MP4 file with H.264 compression using *ffmpeg* as is done in the original Loom implementation [15]. While H.264 is a lossy compression standard, the user should not see any noticeable visual degradation on playback of the archives. Lossless compression algorithms or video file formats can be used as well. Table 1 details the resulting file sizes of our technique using H.264 compression.

3.3.4 Harvesting The DOM

The RAIV Recorder is implemented as a Chrome Extension. When a user opens the recorder, the extension injects code into the active webpage; this allows us to harvest the DOM in the recording process. We implement the following features.

Widget Bounding Box Recognition. We can pinpoint the DOM element the user interacted with when specifying the set of desired interactions to capture. With this, we can automatically detect the bounding boxes of the widgets we will capture. Bounding box recognition addresses a significant limitation of the original limitation; users previously spent much of the recording time manually drawing the bounding boxes of widgets within the dashboards they were capturing. Now that this process is automated, users can spend more time and focus on specifying a more complex series of interactions.

Detecting Widget Type. In addition to automatically detecting widgets, we can use the element type of the targeted widget for further automation. An example of this is in canvas elements. Often, canvases hold a series of widgets that the user may want to iterate through, acting on each of these widgets. Upon an action specification on a canvas, the recorder will detect that the targeted element is a canvas and allow the user to specify the number of times the action should be repeated horizontally and vertically. On capture, the recorder will discretize the canvas and duplicate the action the specified number of times, as shown in Figure 5.

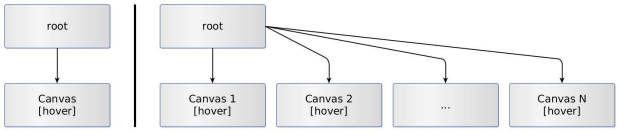

Figure 5: A simplified action tree for a targeted canvas element. A user can specify that any action taken on a canvas element be discretized and repeated N times throughout the entirety of the canvas.

Faster Capture Loop. To minimize the time our UI-bot takes between taking an action and capturing a screenshot, we use Mutation Observers to monitor changes within the document. One limitation of the previous Loom Recorder implementation was that it needed an intelligent concept of when an application has re-rendered after taking an action. Loom set constant timeout to ensure the application's screenshot captured the correct content; this resulted in the reported long capture times. With the DOM, we can monitor for changes in a page resulting from an action. When notified of changes, we trigger the screenshot mechanism.

Manual Capture. Mutation Observers do not handle all cases. If a change to the application is asynchronous, this may fail to capture all changes to the page. As a fallback, users may still specify a wait time on a per-action basis.

For cases that require additional user intervention, we add a feature that we call manual capture. With manual capture, the UI-bot executes the specified action and then waits to take a screenshot until the user gives feedback. In order to give feedback, the user presses the continue recording button that appears in the Chrome extension.

Extracting Metadata. Finally, we can extract attributes from the DOM to collect valuable metadata from the web page on capture. The following sections describe a text-based query system that performs a semantic search on the content embedded within captured web pages. This text is pulled directly from the web page for each node on the action tree after the UI-bot executes an action taken.

Security Implications. The RAIV Recorder Chrome extension conforms to the restrictions enforced by the security policies of Chrome. When installing the RAIV extension, a user will accept and grant permissions to the extension. Given these permissions, the extension will have access to the browser-provided APIs that it needs to function. With user-granted privileges, the RAIV extension can inject a content script into the webpage and screen capture the page. As commonly expected, the RAIV extension only interacts with a web page when directed by the user and is otherwise dormant.

3.4 The RAIV Gallery Server

We have designed RAIV as a standard interface for archived visualizations. We modeled this interface as a YouTube-like sharing platform called the RAIV Gallery. Thus, the RAIV Gallery interface fulfills the objectives of enabling scalability with accessibility (**O2**) and supporting intelligent search capabilities (**O3**).

We anticipate around 10s up to 100s of archived visualizations within the gallery. This poses a new challenge: How can a user find a specific visualization from potentially 5,000 different visual states? Similarly, how can a user find a similar visualization to one they have already seen? This challenge is compounded because many of the frames look similar but not identical, and often, the users do not know what they are searching for.

We have designed the RAIV Gallery to be searchable across two scenarios: i) searching via text; and ii) searching via images.

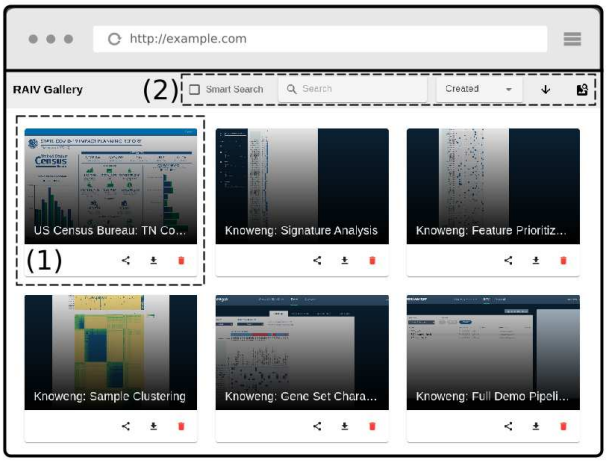

Figure 6: The RAIV Gallery shown in the browser. (1) Every card represents a recorded Loom object. (2) The recordings are filterable and sortable via the navigation toolbar.

The remainder of this section discusses the gallery's design and the search functionalities within the RAIV Gallery.

3.4.1 RAIV Gallery Interface Design

The RAIV Gallery implements a YouTube-like sharing platform for archives where users can instantly access pre-recorded interactive visual insights. The gallery supports two main user types: developers and end users. Developers will specify and capture their desired visual insights, and upon capture, this recording will be uploaded and displayed on the gallery. End users can then navigate to the hosted gallery, find, view, and interact with any previously recorded and uploaded RAIV object. Figure 6 shows the homepage of the gallery.

Creating New Archives. When a user begins capturing their archive, the recorder notifies the gallery and uploads the action tree. For each node within the action tree, the recorder performs the necessary interactions, captures an image of the screen, and uploads this image to the gallery. Once the recorder has finished capturing the tree, it signals the gallery to begin processing and joining the tree and images together. The searchability engine preprocesses the frames during compilation.

Searchability Engine. We have employed a standard technique for relevance filtering based on the AI technologies of Convolutional Neural Networks (CNN) for images and Large Language Models (LLM) for text.

The core idea behind our searchability engine is to transform the representation of text and images into high-dimensional vectors. These vectors represent the meaning behind the text or images and have been found to be compatible with standard distance metrics.

The result is that, instead of the abstract task of "find the image most similar to this one," we can focus on the concrete task of "find the vector closest to this vector in this high-dimensional space."

The searchability engine has three high-level tasks: i) convert each frame into a set of texts and images; ii) process the text and images by an LLM and CNN, respectively; iii) find the closest text and image within the model's embedding space. It's worth noting that the problem of search and query in large image database is not new. Many classic methods were based on similarities or distances [19]. However, more recently embedding space has been shown to outperform [20]. That's why we chose an embedding space based approach. More technical details are in Section 4.2.

Users can utilize the search engine via the navigation toolbar on the RAIV gallery homepage. Enable the smart search option and type directly into the search bar for semantic text search. Click on the image upload button to upload an image for reverse image search. Upon a search, a list of the most similar RAIV objects and their frames will be returned and displayed to the user. The user can further drill down this list by combining the two features.

Provenance View. By default, the playback client has a toggleable overlay that displays a RAIV object's action tree. When the searchability engine recommends a specific object's frame, this overlay reveals the path of actions required to navigate to the frame on the original application. Figure 7 shows an example of this overlay.

4 Results And Discussion

This section evaluates our system architecture. First, we assess the performance of the RAIV Recorder. Then, we evaluate our support for intelligent search capabilities. Finally, we discuss these results.

4.1 Recorder

In Section 3.3, we describe the RAIV Recorder additions to Loom. This section evaluates video size and recording time metrics for seven examples captured by the recorder.

Table 1 shows the Loom video recording storage size and resolution and the number of interactions for seven test cases. The KB/Interaction ratio also shows how the recording increases in size on disk for each interaction. All test cases vary in video size due to RAIV's ability to capture entire web pages rather than what is only in view. The Knoweng test cases show how much a web page's height can vary even within the same application. Interestingly, the Knoweng recordings of the Sample Clustering dashboard are of similar size, but the recording that uses the Canvas action duplication feature has over 1,100 more frames; this is because the frames with repeated canvas actions are similar to one another. The video compression allows us to store these extra frames with minimal cost. We expect most cases where one would use this canvas feature to follow a similar behavior.

Table 2 compares recording specification times for six of the seven examples. Specification time typically varies depending on the complexity of the interactions a user records. Compared to Loom, users spend less time specifying the exact widgets they want to capture, and instead, complex sequences of interactions take up most of a user's time. Recordings in this table with more significant specification times have action maps that often have trees of greater depth. Capture time changes based on two factors. First, if the number of actions the user specifies is large, the capture time will grow linearly. Second, if the application consists of asynchronous DOM updates, the user will use the set wait

Figure 7: Provenance view on playback. When users select a recommended frame in a RAIV Recording from the gallery view, they can toggle on an overlay showing the object's action tree. A user can select any action node and see the corresponding frame. This view shows the recommended action (yellow) and the path a user can take to get to the recommended action (red).

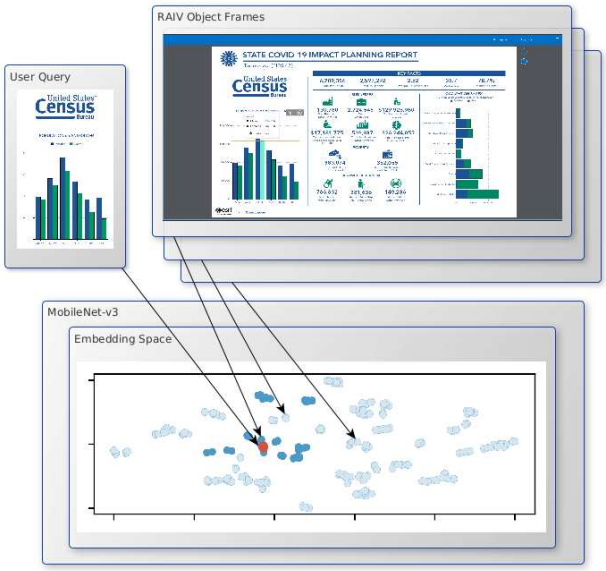

Figure 8: Example of querying via reverse image search. The embedding space shown is a TSNE projection of objects within the vector database.

time for screenshots or the manual capture feature. Both of these options will increase capture time. The US Census Bureau: TN Covid-19 Impact Planning Report exemplifies this behavior.

4.2 Searchability Engine

Section 3.4 describes an engine for performing relevance filtering across all archived visualizations. In this section, we define the specific models and techniques RAIV employs.

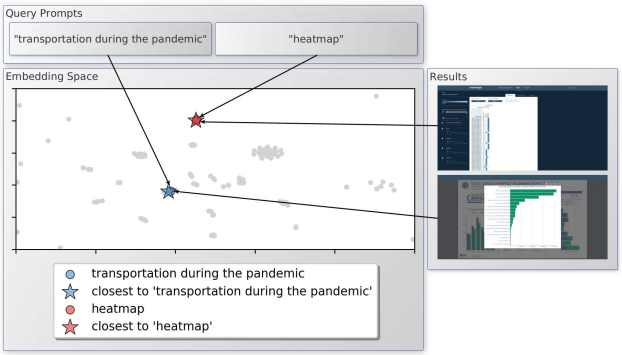

Figure 9: Semantic text search results. Text queries input via the search bar in the gallery view are fed through an LLM to get an embedding representative of the meaning of the text. These embeddings are then compared to embeddings from the text extracted from the DOM upon capture. This figure shows two example queries and their corresponding recommended RAIV view.

Text. To generate the dataset of text embeddings, we use the text scraped from the captured page for each action. We clean the scraped text by converting all text to lowercase, removing all non-character values, removing stopwords, and lemmatizing each token. We also add the user-specified title for the archive and the web page's title to the text. We refer to the cleaned text as the frame's "tags."

Image. The nature of CNN models for images is that they are resilient to changes in the images. As a result, we can use the entire frame of the application as the input to our model, with effectively no preprocessing.

Models. For text embedding, we use a small BERT model called `all-MiniLM-L6-v2`; this model is 100 MB in size and

Source Application	#I	Size (MB)	Resolution	(MP)	KB/I
US Census Bureau: TN Covid-19 Impact Planning Report	39	0.7	1900×944	(1.7)	19
Knoweng: Full Demo Pipeline	33	1.8	1900×2652	(5.0)	55
Knoweng: Signature Analysis	5	0.4	1886×1714	(3.2)	73
Knoweng: Gene Set Characterization	78	0.3	1900×698	(1.3)	3.4
Knoweng: Sample Clustering	65	4.4	1886×2716	(5.1)	70
Knoweng: Sample Clustering [Canvas (20x20)x3]	1203	5.2	1885×2716	(5.1)	4.4
Knoweng: Feature Prioritization	25	4.6	1886×7824	(15)	190

Table 1: Summary of the sizes of archived objects compared to the number of interactions (#I).

	Specification Time (m:s)			Capture Time (m)		
Source Application	min	avg	max	min	avg	max
US Census Bureau: TN Covid-19 Impact Planning Report	3:06	3:39	4:01	0:48	0:49	0:50
Knoweng: Full Demo Pipeline	*	*	*	*	*	*
Knoweng: Signature Analysis	0:14	0:16	0:18	0:15	0:15	0:15
Knoweng: Gene Set Characterization	0:33	0:38	0:43	0:34	0:34	0:35
Knoweng: Sample Clustering	3:43	4:40	6:52	1:17	1:21	1:26
Knoweng: Sample Clustering [Canvas (20x20)x3]	0:40	0.44	0:51	7:24	7:25	7:26
Knoweng: Feature Prioritization	1:17	1:21	1:23	1:35	1:38	1:40

Table 2: Summary of the time to capture archived objects. *Not collected due to time constraints.

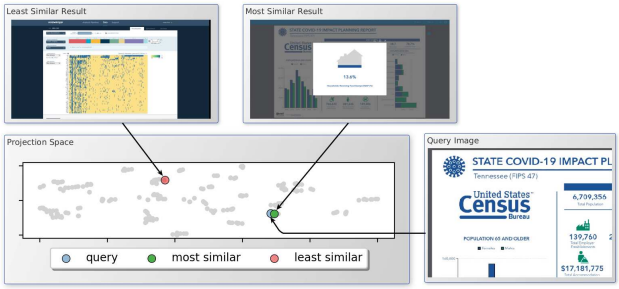

Figure 10: Reverse image search results. Image queries input via the image search dialogue in the gallery view are fed through a CNN to get a representative embedding. This embedding is then compared to the embeddings of all stored RAIV object frames. This figure shows an example image query and the most and least similar results.

transforms around 1 KB of text into a 384-dimensional vector. We chose this model as it provides generalized semantic text embeddings on a CPU with minimal latency for search and lookup. For image embedding, we use a small `MobileNetV3` model (11 MB) that transforms an RGB image into a 1024-dimensional vector. We chose this model because its outputs generalize across the image space well due to having fewer parameters, limiting the risk of over fitting on the image. We have found additional benefits to using a small model, namely that it runs more efficiently on affordable cloud instances.

Retrieval. After the tags and images have been transformed into high-dimensional vectors, these vectors get stored within an open-source embeddings database called *ChromaDB*. Using Python, we can query ChromaDB for other high-dimensional vectors with the smallest distance from a query vector. For example, Figure 10 illustrates three of the embeddings of three images and how the distance between them correlates with their similarity in image space. For distance calculations, we use cosine similarity.

4.3 Discussion

Currently RAIV targets only web based visualizations because harvesting elements from within a web app is already a common practice, for instance in the cybersecurity research community. It requires significant future work to develop a portable RAIV library for all desktop or even mobile app developers to adopt, such that future visualization applications can include a capturing functionality as a native part of their application.

The work to develop intelligent search capabilities of RAIV in this paper assumes that large archives of interactive visualizations will exist. That is not the reality currently, hence the examples shown in this work are only preliminary prototypes. A key research gap that needs to be filled is how provenance data should be integrated into such large archives along with what real-world use cases these future archives need to serve.

The current web-based focus of RAIV leads to another important challenge. That is, the vast and fast-paced creativity in the web space, especially because web developers are not bound by standard UI interactions, they can develop new UI widgets using customized Javascript libraries at will. Even though RAIV covers all of the widget types used by KnowEnG, US Census and Tableau, it is important to note that RAIV does not cover all UI widgets known to us and probably will not ever in the future because of the boundless creativity of the open source community. As some current limitations, RAIV does not support widgets with non-rectangular bounding boxes, widgets embedded within iframes, and widgets with asynchronous behaviors.

5 Conclusion

This paper presents RAIV as an end-to-end archival service for recording and sharing web-based interactive visualizations without sharing the underlying data or software. We showed examples of the RAIV recorder capturing complex sequences of interactions in actual analytic pipelines. On the capturing front, we im-

proved upon past work by allowing developers and data analysts to create recordings of visual analytics in less time with greater ease. The Chrome extension-based recorder allows for a better level of adoption. The supported user actions include: clicking, hovering, toggling, and sliding.

The RAIV gallery server is a novel platform for sharing interactive visualization archives without manual work from the developer. RAIV enhances transparency and allows a more expansive base of users to delve into shared analytics to gain valuable insights that, previously, may not have been shareable or discoverable. We have also shown that these archives can be indexed on a frame level via multiple modalities, effectively searching through a crowd-sourced analytics database.

A core contribution in this work is the creation of a Searchability Engine that allows users to selectively filter their collection of archived visualizations for relevant interactions and visual outputs. In contrast with common approaches to image similarity that are very specific to the problem domain they target, we have found great success in the application of general-purpose foundational AI models, completely untargeted for our problem domain of archived visualization applications.

Lastly, with the recent progress of general-purpose AI and related to RAIV, we feel large archives of interactive visualization will provide many research opportunities that may be unfathomable today. For better adoption and increased longevity of the platform, RAIV will be available as open source software.

6 Acknowledgments

The authors would like to thank the anonymous reviewers of this and previous versions of the manuscript for their valuable comments and suggestions. The authors are supported in part by NSF Award IIS-2209767 and CCRI-1925615.

References

[1] D. Atkins, T. Dietterich, T. Hey, S. Baker, S. Feldman, and L. Lyon. Final Report: National Science Foundation Advisory Committee for Cyberinfrastructure Task Force on Data and Visualization, 2011.

[2] F. Berman and R. Rutenbar. Realizing the Potential of Data Science - Final Report from the National Science Foundation Computer and Information Science and Engineering Advisory Committee Data Science Working Group, 2016.

[3] F. L. Cook. NSF 18-053: Dear Colleague Letter: Achieving New Insights through Replicability and Reproducibility. 2018.

[4] J. Crocker and M. L. Cooper. Addressing Scientific Fraud. *Science*, 334(6060):1182–1182, dec 2011. doi: 10.1126/science.1216775

[5] J. Freire. Making computations and publications reproducible with vistrails. *Computing in Science & Engineering*, 14(4):18–25, 2012.

[6] D. Koop, E. Santos, P. Mates, H. T. Vo, P. Bonnet, B. Bauer, B. Surer, M. Troyer, D. N. Williams, J. E. Tohline, et al. A provenance-based infrastructure to support the life cycle of executable papers. *Procedia Computer Science*, 4:648–657, 2011.

[7] J. Kurose. NSF 17-022: Dear Colleague Letter: Encouraging Reproducibility in Computing and Communications Research. 2016.

[8] J. Lasser. Creating an executable paper is a journey through open science. *Communications Physics*, 3(1):143, 2020.

[9] C.-U. Lim, R. Baumgarten, and S. Colton. Evolving behaviour trees for the commercial game defcon. In *European Conference on the Applications of Evolutionary Computation*, pp. 100–110. Springer, 2010.

[10] S. Malcomber, M. Martonosi, J. Moore, S. Marguilies, A. Isern, A. Knoedler, K. Sharp, S. Jones, K. Craig-Henderson, and E. Gianchandani. NSF 23-018: Dear Colleague Letter: Reproducibility and Replicability in Science. 2023.

[11] M. McNutt. Reproducibility. *Science*, 343(6168), 2014.

[12] J. P. Mesirov. Accessible reproducible research. *Science*, 327(5964):415–416, 2010.

[13] T. Miyakawa. No raw data, no science: another possible source of the reproducibility crisis. *Molecular brain*, 13(1):1–6, 2020.

[14] J. M. Perkel. Data visualization tools drive interactivity and reproducibility in online publishing. *Nature*, 554(7690):133–134, jan 2018. doi: 10.1038/d41586-018-01322-9

[15] M. Raji, J. Duncan, T. Hobson, and J. Huang. Dataless sharing of interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(9):3656–3669, 2020.

[16] I. Santana-Perez, R. F. da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández, and O. Corcho. Reproducibility of execution environments in computational science using semantics and clouds. *Future Generation Computer Systems*, 67:354–367, 2017.

[17] N. Springer. Reality Check on Reproducibility. *Nature*, 533(7604):437–437, May 2016. doi: 10.1038/533437a

[18] P. B. Stark. Before reproducibility must come preproducibility. *Nature*, 557(7706):613–614, 2018.

[19] C. Wang, J. P. Reese, H. Zhang, J. Tao, Y. Gu, J. Ma, and R. J. Nemiroff. Similarity-based visualization of large image collections. *Information Visualization*, 14(3):183–203, 2015.

[20] Y. Ye, R. Huang, and W. Zeng. Visatlas: An image-based exploration and query system for large visualization collections via neural image embedding. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–15, 2022. doi: 10.1109/TVCG.2022.3229023

7 Author Biography

Hunter Price received his BS in Computer Science from the University of Tennessee, Knoxville, where he is currently an MS student. His research interests include machine learning, data visualization, and intelligent systems.

John Duggan received his B.S. and M.S. in Computer Science from the University of Tennessee, Knoxville, where he currently works as a Research Software Engineer.

Robert Sisneros is a Senior Research Scientist at the National Center for Supercomputing Applications. His research covers visualization, data model, parallel algorithm, I/O optimization, and big data. He earned BS in Mathematics and Computer Science from Austin Peay State University, and MS and PhD in

Computer Science from the University of Tennessee, Knoxville.

James Hammer received BS from the University of Tennessee Knoxville in Computer Science, where he is currently a PhD student. His research interests include real-time rendering, data visualization systems, and visualization architectures.

Tanner Hobson is a research scientist at the University of Tennessee, where they also received their Ph.D. in Computer Science in 2023. Their research interests include visualization web service, artificial intelligence, and cloud computing.

James Osborne is a researcher at the University of Tennessee, where he also earned his BS in Computer Science. His interests include operating systems, open source, and frameworks for remote data analysis and visualization.

Jian Huang is a professor in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. His research focuses on data visualization and analytics. He received his PhD in Computer Science from the Ohio State University in 2001. His research has been funded by NSF, Department of Energy, Department of Interior, NASA, UT-Battelle, and Intel.