# IMAGE: A Deployment Framework for Creating Multimodal Experiences of Web Graphics

Juliette Regimbal
juliette.regimbal@mail.mcgill.ca
McGill University
Montréal, Québec, Canada

Jeffrey R. Blum
jeffrey.blum@mail.mcgill.ca
McGill University
Montréal, Québec, Canada

Jeremy R. Cooperstock
jer@cim.mcgill.ca
McGill University
Montréal, Québec, Canada

## ABSTRACT

Existing screen reader software can convey graphical content to blind and low vision web users through text information, but does not offer richer multimedia representations. Standalone research projects have attempted to fill this gap, but have not achieved lasting, widespread deployment, thus motivating the creation of a common platform for implementing and deploying multimodal experiences. We are creating the IMAGE system to be an open-source "playground" for prototyping, exploring, and deploying novel solutions to accomplish this. IMAGE does not replace a screen reader or alt-tags, but rather works with them to provide a more complete understanding of web graphics. In this communication, we describe how the IMAGE browser extension and server components form a modular, extensible system that can accelerate the development of new haptic and audio renderings. We explain how various parties, whether as developers or designers, can benefit from this architecture, building on our pipeline for their own purposes.

## CCS CONCEPTS

• **Human-centered computing → Accessibility systems and tools**; *Interactive systems and tools*; • **Software and its engineering → ** *Software libraries and repositories.*

## KEYWORDS

web graphics, open source, software framework, multimodality

**ACM Reference Format:**
Juliette Regimbal, Jeffrey R. Blum, and Jeremy R. Cooperstock. 2022. IMAGE: A Deployment Framework for Creating Multimodal Experiences of Web Graphics. In *19th Web for All Conference (W4A'22), April 25–26, 2022, Lyon, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3493612.3520460

## 1 INTRODUCTION

Reliance of blind and low-vision users on alternative text (alt-text) image descriptions is problematic. Although the HTML Standard states "the intent [of alternative text] is that replacing every image with the text of its alt attribute does not change the meaning of the page" [1], in practice, this may be challenging. For example, text descriptions for information-dense images such as a chart of COVID-19 case data, are unlikely to support the same engagement with, or access to, the information a sighted user would have. Moreover, alternative text is not provided for all images, and even when it is, the associated information is often not useful. For example, Gleason et al. found that of the few images posted by users on Twitter that contained an image description, 39.9% were rated as either "irrelevant" or "somewhat relevant" to the image itself [5].

The Internet Multimodal Access to Graphical Exploration (IMAGE) project aims to improve the accessibility of such images and graphics by processing them to produce rich audio and haptic outputs on demand, in addition to spoken text description, that can be navigated by most screen readers. IMAGE supports a variety of possible inputs—e.g., maps, photographs, line charts—and possible outputs—e.g., spatialized non-speech audio, speech audio, force-feedback haptics.

To support the rapid development of the components necessary to transform a graphical input to semantically relevant audio and haptic outputs, we are implementing an open-source software architecture. Our objective is for this architecture to allow for quick prototyping and redesign of the entire multimedia pipeline, features often missing in similar projects. Previous systems for generating accessible outputs from visual inputs, such as VizWiz [3], connect software modules in an ad hoc method that limits modification and reuse, especially by those uninvolved with the initial project. In contrast, the generic architecture and associated components implemented as part of IMAGE permit flexible development and can be used in other projects that aim to transform arbitrary input media to a set of output media.

Given the pre-release status of the architecture and flexibility of its components, we do not discuss end-user evaluation of the actual renderings produced by the system. Instead, the primary contribution of this communication is a first look at IMAGE as a platform, specifically an open source, modular framework that enables researchers and developers to more easily implement, test, and deploy new methods for generating multimodal audio and haptic experiences, especially for those who are blind or low vision. We discuss the key design decisions that led to the IMAGE architecture, and provide an overview of the architecture with examples of how one would implement their own functionality within it. Further technical details are available in the IMAGE open source code repositories, linked in this article.

## 2 BACKGROUND

Various projects similar to IMAGE follow a three-step pipeline consisting of collecting data, whether from sensors or a media file, processing the data to extract usable information, and using the information to synthesize a new output in another modality or modalities. In Table 1, five projects are examined in the context of this pipeline.

Twitter A11y generates alternative text for inaccessible images on Twitter, applying different methods depending on the type of image encountered: URL previews, text, or neither [7]. Users of VizWiz use their smartphone's camera to capture an image of an object of interest. They are then able to ask questions about the object and receive crowdsourced answers through the application itself [3]. Winters et al.'s proposed pipeline would create rich sonifications using a variety of data available on a social media site [10]. The JAWS screen reader includes the Picture Smart service to generate a description for an image submitted by a user [9]. The VoiceOver Recognition feature produces descriptions from images on an iPhone and allows navigation of the elements within them [6].

All the aforementioned projects are either defunct (VizWiz and Twitter A11y), proposals (Winters et al.'s pipeline), or proprietary software (Picture Smart and VoiceOver). The present dearth of available resources therefore poses a challenge for researchers and developers aiming to create or build upon existing pipelines in this space. Accordingly, implementation of a new framework is needed, for which reusable components are highly desirable. The utility of well-documented, reusable components that can be used to create new systems has long been understood in software engineering [4]. Additionally, as modules become more self-contained and require less specialist knowledge to integrate into a system, designers are able to prototype with more creativity and perceived efficiency as evidenced through the "maker movement" [8]. A set of modules suitable for the task of converting graphical content into non-visual multimodal outputs and a framework to easily arrange them would be a useful tool to these researchers and developers.

## 3 ARCHITECTURE

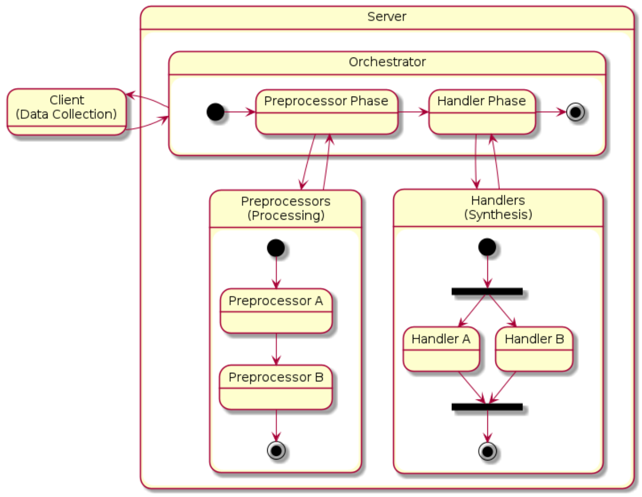

**Figure 1: A high-level diagram of the software architecture showing how it models the three-step pipeline. The preprocessors run serially while the handlers run in parallel.**

The IMAGE architecture uses a server-client model and explicitly supports the three-step pipeline described in Section 2. The data collection step is accomplished on the client side, in the current implementation of IMAGE, via a Google Chrome browser extension. At this point, the data collected is typically an image file encountered while browsing, but may also be a geographic coordinate representing a point of interest displayed on an embedded Google map. Additional information, such as the context of the web page the image data was taken from, user preferences, and client capabilities are also sent from a client to the server. These elements are discussed further in Section 3.1.

The processing step is represented by a set of microservices on the server called *preprocessors*. These receive HTTP POST requests containing the information from the client and perform a specific task—e.g., applying a machine learning model, calling a third-party API—to obtain more useful information about the web graphic in question. The outputs of these preprocessors can also be combined, allowing a preprocessor to take advantage of the output of any earlier stage.

The synthesis step is performed by another set of microservices called *handlers*, which receive the entire set of information from the preprocessors and all information collected by the client. Each handler then determines if it is capable of producing a usable output—a *rendering* in IMAGE terminology—and creates it using built-in technologies and *services* that perform functions (e.g., text-to-speech) that are generic to multiple handlers. Multiple renderings for a single graphic may be produced by the handlers, providing different perspectives.

Communications between the components carrying out these three steps are executed by an orchestrator microservice. This communicates with the client, schedules and performs all communication with preprocessors and handlers, and assembles the final collection of renderings synthesized for the chosen graphic. The content of all communications must validate against the JSON schema(s) describing that message. All IMAGE microservices are open source Docker images available from https://github.com/Shared-Reality-Lab/IMAGE-server. The browser extension is available from https://github.com/Shared-Reality-Lab/IMAGE-browser.

### 3.1 Client Requests, Capabilities, and Renderers

The requests sent from the client include base generic information regardless of the type of graphic being queried. All requests include a Universally Unique Identifier (UUID), an ISO 639-1 representation of the desired output language, the time the request was made in Unix time, serialized XML representing the node containing the graphic, and sets of capabilities and renderers. From there, additional fields are added based on the media type; an image request also contains the file in a base64 encoding and the dimensions of the file as displayed on the page in pixels.

The sets of capabilities and renderers, each containing reverse domain name identifiers, indicate respectively the desired media to be returned, and the ability of the client to display encodings of renderings.

The capabilities indicate the preferences or user needs—e.g., to receive spatialized audio as an output—and relevant hardware peripherals available. This latter point is especially important for

| Project | Data Collection | Processing | Synthesis |
|---|---|---|---|
| Twitter A11y [7] | Images | Crowdsourcing, OCR, URL following | Text description |
| VizWiz [3] | Smartphone camera | Crowdsourcing | Responses to questions |
| Winters et al. [10] | Social media elements | Microsoft APIs | Context-specific sonifications |
| Picture Smart [9] | Images | OCR, Object detection, etc. | Text description |
| VoiceOver Recognition [6] | Images | OCR, Object detection, etc. | Description, interactive exploration |

**Table 1: Previous projects and how each fits the pipeline discussed in Section 2.**

haptic renderings, since force-feedback devices and raised pin arrays are not commonly owned and lack standard communication protocols. The advertised capabilities are intended to ensure that the media content returned can be properly displayed to and perceived by the user.

In the context of IMAGE, the renderers are data structures specifying how to present a rendering in software. For example, an audio rendering, made up of different sections that can be presented independently, would need to convey the audio data and the location of each section in the audio data. A particular format for this in IMAGE is indicated by the `ca.mcgill.a11y.image.renderer.SegmentAudio` identifier. A client that includes this identifier in the request indicates that it is capable of parsing and displaying the data received in this format, which is also specified using a JSON schema.

## 3.2 Orchestrator

The orchestrator microservice is the only server component that directly communicates with the client. Upon receiving the request and checking to ensure it is in a valid, understood format, the orchestrator queries for a list of available preprocessors and handlers. This is done by checking for running containers that contain either the preprocesor Docker label (`ca.mcgill.a11y.image.preprocessor`) or the handler Docker label (`ca.mcgill.a11y.image.handler`) and have at least one network in common with the orchestrator. As this query is performed for each request, preprocessors and handlers can be added or removed at runtime.

It then orders the preprocessors following their advertised priority (see Section 3.3) and serially forwards the client request with preprocessor output data to each. Each preprocessor has a fixed amount of time to respond—in IMAGE this is currently set to 15 seconds—before the orchestrator aborts the request.[1] If the preprocessor does respond in the allotted time, it either provides a key and the data generated that the orchestrator will append to a `preprocessors` key in the client request, or else it responds with a “204 No Content” HTTP status to indicate that it is not able to supply more information.

After the preprocessors have run, the orchestrator then forwards the client request, including all the preprocessor outputs, to the handlers in parallel. The handlers are then expected to respond with an array containing the list of the renderings, if any, that were produced. The renderings are concatenated and returned to the client to be displayed to the user.

[1]In practice, total user response times for the entire pipeline are generally 3–10 s.

## 3.3 Preprocessors

Each preprocessor must respond to an HTTP POST containing the request. The preprocessor then performs its task to extract a certain kind of data. If successful, it replies with a key indicating the kind of data returned and the actual content as a JSON object. This key is not tied to a particular preprocessor microservice. For example, many different machine learning object models exist and one may be better suited to a particular task than another. If the preprocessor microservices that run each model both use the same data structure (e.g., `ca.mcgill.a11y.image.preprocessor.objectDetection`), then any later component will still be able to correctly parse the output of either. This allows preprocessors to be used as drop-in replacements of each other so long as they use the same data structure.

As previously mentioned, preprocessors run serially and indicate a priority at which to run. This priority is communicated to the orchestrator by numeric value in the Docker label. For example, a preprocessor would be in the first priority group by having the label `ca.mcgill.a11y.image.preprocessor: 1`. Since each request has previous successful results added to it, preprocessors in later groups are guaranteed to obtain the outputs of those in early groups that ran successfully. This permits two cases where preprocessors use the outputs of others. In one case, a preprocessor requires previous results to generate a response of its own. For example, a preprocessor that groups the elements already found by an object detection algorithm cannot run without any objects. As such, if the object detection preprocessor does not run, the grouping preprocessor must return a 204 status.

In the second case, a preprocessor can use information from a previous preprocessor to choose not to run if it would be a waste of time and resources. For example, that same object detection preprocessor may be designed to work only on photographs. In the absence of other information, this preprocessor should run on all inputs, since it does not know whether the graphic is indeed a photograph, and will have to make its own determination whether it can effectively treat the content. However, if a previous preprocessor has determined that the graphic is definitely not a photograph, e.g., a chart, then the object detection preprocessor can safely return a 204 status as its results would be irrelevant.

## 3.4 Handlers and Services

The handlers each receive a POST request containing the data from the client and from the preprocessors that ran successfully. Each handler must then determine which renderings, if any, it can generate. This is based on the type of data included in the request (e.g., image file), the advertised capabilities and renderers, and the returned preprocessor data. Any supported renderings are then generated by whichever method is appropriate.

For example, a handler that focuses on objects in a photograph can generate a text description of the detected attributes (e.g., indoor or outdoor, type of location or focus) and the detected objects. This text can be returned as a rendering that may be used by a built-in screen reader or Braille display. The same text can be processed through a text-to-speech engine and then spatialized using higher-order ambisonics, a method of surround audio calculated using sources positioned around the listener [2]. Pings representing the locations of the objects can be interspersed, and the result returned as a rendering in the form of an audio file. Offsets for each semantic part of the file—i.e., the introduction and each object or group of objects—can be generated as well, and used to create a third rendering supporting easy user navigation within the audio file.

Certain tasks, such as text-to-speech (TTS) or generating higher-order ambisonic audio, are used across many handlers and sufficiently specialized to make implementing them in each handler an onerous task. This motivates the concept of *services* to support the generation of renderings in handlers. These services exist in their own microservices and provide a separate API to enable many handlers to perform these kinds of tasks. For example, the TTS service currently used in IMAGE receives an HTTP POST request containing the set of words or phrases to be processed. It then responds with an audio file containing the resulting synthesized speech and the location of each word or phrase in the file for further processing. This audio file and metadata are then sent to a SuperCollider service that, when triggered with certain Open Sound Control (OSC) commands, generates a new audio file containing speech and non-speech sound, spatialized using ambisonics.

## 4 DISCUSSION

To better illustrate how the IMAGE architecture supports flexible development and deployment, we consider two use cases.

First, we consider a designer wishing to integrate a new spatialized sound rendering of the contents of photographs. If existing preprocessors provide sufficient information for the designer's rendering, then they need only create a new handler that follows one of the existing renderer data formats. This can be in any language or tools in which the designer is comfortable, so long as the eventual Docker container encapsulating the new functionality follows the simple data exchange format. These portions can be taken from templates available in the source repository, further lightening the burden. The designer need not make any changes to the browser extension, preprocessor chain, or any other handlers. Indeed they do not even need to recompile the entire system, but only create their new handler in a Docker container, which can then be dynamically inserted into an already running server by editing a single configuration file. In this scenario, the IMAGE architecture's flexibility due to formally defined data formats and modular microservices means that the designer need only work within the strict confines of their handler container, with complete freedom inside that boundary.

Next, we consider a machine learning researcher developing an improved object recognizer. Similar to the designer scenario above, the researcher can create a preprocessor container that uses the same inputs and outputs as the default object detection container, and simply swap it out. Once done, it can be tested by end users via the browser extension they already have installed, with the handlers creating renderings based on the new object detector. This provides researchers with a practical testbed and deployment opportunity that would formerly require building up the complete system from scratch, an often insurmountable goal. By simplifying the path from research prototype to deployment, we hope that more machine learning researchers will consider offering their work for practical use by the blind and low vision community.

## 5 FUTURE WORK

IMAGE and its architecture are works in progress. Although the software structure has been successful so far in the development of IMAGE, we hope that with the open source release of its software, it may be used by others contributing to our project or in other unrelated projects. As this occurs, we intend to refine the architecture described here in response to the experiences of users. Optimizations for performance will also be necessary as IMAGE scales with more users and possibly needs to run across multiple systems.

Additionally, at this point the architecture of IMAGE and its components are more opaque than is ideal. High-quality documentation and tools to visualize the state of a system running with IMAGE's architecture would be notable improvements to the usability of the components discussed in this paper and others in the IMAGE project. These would also aid quantitative evaluation of a running system (e.g., maximum response time). Improvements in documentation should make it easier for other developers to get started. Similarly, improved debugging and visualization tools should aid them when they encounter problems.

## 6 CONCLUSION

We introduced the Internet Multimodal Access to Graphical Exploration (IMAGE) project, along with its open-source framework, designed to handle the three phases common to processing graphical information into outputs accessible to users who are blind or low vision. These outputs can range in complexity from plain text to interactive multimodal experiences. The IMAGE architecture, formed of many independent microservices that are synchronized by an orchestrator component, is highly modular and designed to support quick development within individual components and over the system as a whole. As the project progresses and the framework discussed here becomes more mature through the addition of new features and evaluation, we hope it will serve as a useful resource for researchers and developers who create multimodal outputs from inaccessible source media. Those interested in IMAGE can learn more about the project through the repositories linked above or the project site at https://image.a11y.mcgill.ca.

## ACKNOWLEDGMENTS

IMAGE is funded by the Accessible Technology Program of Innovation Science Economic Development Canada, project #513221. The authors wish to acknowledge the numerous designers and developers who contributed to the many components within the IMAGE architecture.

## REFERENCES

[1] 2022. *HTML Standard: Images*. Technical Report. Web Hypertext Application Technology Working Group (WHATWG). Retrieved 2022-01-07 from https://html.spec.whatwg.org/multipage/images.html#alt

[2] Joseph Lloyd Anderson. 1999. What is Ambisonics and how can I get some. *eContact!* 2.4, 9 (1999). https://econtact.ca/2_4/Ambisonics.htm

[3] Jeffrey P. Bigham, Chandrika Jayant, Andrew Miller, Brandyn White, and Tom Yeh. 2010. VizWiz::LocateIt - enabling blind people to locate objects in their environment. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*. IEEE, San Francisco, CA, USA, 65–72. https://doi.org/10.1109/CVPRW.2010.5543821

[4] G. Fischer. 1987. Cognitive View of Reuse and Redesign. *IEEE Software* 4, 4 (July 1987), 60–72. https://doi.org/10.1109/MS.1987.231065 Conference Name: IEEE Software.

[5] Cole Gleason, Patrick Carrington, Cameron Cassidy, Meredith Ringel Morris, Kris M. Kitani, and Jeffrey P. Bigham. 2019. "It's almost like they're trying to hide it": How User-Provided Image Descriptions Have Failed to Make Twitter Accessible. In *The World Wide Web Conference on - WWW '19*. ACM Press, San Francisco, CA, USA, 549–559. https://doi.org/10.1145/3308558.3313605

[6] Apple Inc. 2021. Use VoiceOver for images and videos on iPhone. Retrieved 2022-01-17 from https://support.apple.com/en-ca/guide/iphone/iph37e6b3844/ios

[7] Christina Low, Emma McCamey, Cole Gleason, Patrick Carrington, Jeffrey P. Bigham, and Amy Pavel. 2019. Twitter A11y: A Browser Extension to Describe Images. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, Pittsburgh PA USA, 551–553. https://doi.org/10.1145/3308561.3354629

[8] Joel Sadler, Lauren Shluzas, Paulo Blikstein, and Riitta Katila. 2016. Building Blocks of the Maker Movement: Modularity Enhances Creative Confidence During Prototyping. In *Design Thinking Research: Making Design Thinking Foundational*, Hasso Plattner, Christoph Meinel, and Larry Leifer (Eds.). Springer International Publishing, Cham, 141–154. https://doi.org/10.1007/978-3-319-19641-1_10

[9] Freedom Scientific. 2019. What's New in JAWS 2019 Screen Reading Software. Retrieved 2022-01-07 from https://support.freedomscientific.com/Downloads/JAWS/JAWSWhatsNew?version=2019

[10] R. Michael Winters, Neel Joshi, Edward Cutrell, and Meredith Ringel Morris. 2019. Strategies for Auditory Display of Social Media. *Ergonomics in Design* 27, 1 (Jan. 2019), 11–15. https://doi.org/10.1177/1064804618788098 Publisher: SAGE Publications Inc.