PDF Download
3604258.pdf
30 March 2026
Total Citations: 1
Total Downloads: 953

Latest updates: https://dl.acm.org/doi/10.1145/3604258

Published: 13 September 2023

Citation in BibTeX format

RESEARCH-ARTICLE

Frappé: An Ultra Lightweight Mobile UI Framework for Rapid API-based Prototyping and Environmental Deployment

ADIL RAHMAN, University of Virginia, Charlottesville, VA, United States

SEONGKOOK HEO, University of Virginia, Charlottesville, VA, United States

Open Access Support provided by:

University of Virginia

Proceedings of the ACM on Human-Computer Interaction, Volume 7, Issue MHCI (September 2023)
https://doi.org/10.1145/3604258
EISSN: 2573-0142

Frappé: An Ultra Lightweight Mobile UI Framework for Rapid API-based Prototyping and Environmental Deployment

ADIL RAHMAN, University of Virginia, USA
SEONGKOOK HEO, University of Virginia, USA

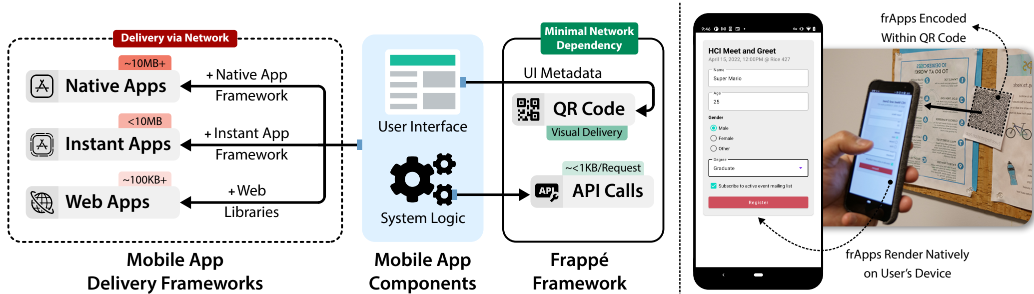


Fig. 1. QR codes are typically limited in their interactive capabilities and can only redirect users to other sources. Frappé augments the interactive capabilities of QR codes, allowing them to directly serve ultra lightweight and functional mobile UIs.

QR codes have been used as an inexpensive means to connect users to digital platforms such as websites and mobile applications. However, despite their ubiquity, QR codes are limited in purpose and can only redirect users to the URL contained within it, thereby making their use heavily network dependent, which can be unsuitable for use in ephemeral scenarios and areas with limited connectivity. In this paper, we introduce *Frappé*, a framework capable of deploying ultra lightweight UIs to mobile devices directly through QR codes, without requiring any network connectivity. This is achieved by decomposing the UI into metadata and storing it inside the QR code, while offloading the UI functionality to API calls. We also introduce *enFrappé*, a WYSIWYG tool for building Frappé UIs. We demonstrate the lightweight nature of our framework through a technical evaluation, whereas the usability of our UI builder tool is demonstrated through a user study.

CCS Concepts: • **Human-centered computing** → **User interface toolkits**.

Additional Key Words and Phrases: Lightweight Mobile UI Framework; Rapid Application Prototyping; Mobile User Interface; Functional Prototyping; Mobile UI Builder

ACM Reference Format:
Adil Rahman and Seongkook Heo. 2023. Frappé: An Ultra Lightweight Mobile UI Framework for Rapid API-based Prototyping and Environmental Deployment. *Proc. ACM Hum.-Comput. Interact.* 7, MHCI, Article 211 (September 2023), 23 pages. https://doi.org/10.1145/3604258

1 INTRODUCTION

With the proliferation of smart devices, almost everything we interact with has adopted a digital presence. There is a mobile app for everything we interact with, augmenting these interactions with

Authors' addresses: Adil Rahman, University of Virginia, Virginia, USA, adil@virginia.edu; Seongkook Heo, University of Virginia, Virginia, USA, seongkook@virginia.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2023 Copyright held by the owner/author(s).
2573-0142/2023/9-ART211
https://doi.org/10.1145/3604258

a unique digital experience. However, such experiences come with the overhead of downloading and installing the apps, often creating a usability barrier [9, 47]. Web apps have been the most obvious choice for addressing this usability barrier since they can be loaded directly on a browser without requiring any installation [12, 47]. Moreover, Google and Apple recently introduced their own mini-app frameworks, *Instant Play* [17] and *App Clips* [24], respectively, which allow developers to create a minified version of their full-scale applications that can be launched directly on smartphones. There also exist services that allow users to stream applications directly over the cloud to their personal devices [39, 40, 46], or download app slices based on their requirement [9]. However, while all of these methods allow users to access apps without the overhead of downloading and installing them, they are heavily dependent on the network connectivity for loading the apps which may limit their ubiquity. Furthermore, these techniques might pose a usability barrier if there is limited network connectivity, or if the app or service is ephemeral in nature, e.g., one-time use. For instance, if one is traveling to remote areas where there might be limited network connectivity, such as national parks or other countries, accessing these apps can be challenging. Thus, there needs to be an interaction medium to support such ephemeral use cases in areas with limited network connectivity.

QR codes have been used as an inexpensive means for connecting users to digital platforms [3, 27]. However, QR codes have been typically associated with a one-dimensional usage – storing URLs to redirect users to other sources such as websites and mobile apps [3] – owing to their limited storage capacity. However, despite this, QR codes have several positive attributes – they do not require network connectivity to be read; they are easily accessible; contactless; cheap to produce; and ubiquitous. Furthermore, another important characteristic of QR codes is its locality – the placement of QR code can be directly associated with its use case. For example, if there is a QR code on a smart light, it can be inferred that the QR code contains the URL for the associated smart light controller app. However, despite these attributes, typical QR code interactions today are static and rely heavily on network connectivity.

In this paper, we present *Frappé* [1] , a framework for deploying ultra lightweight functional mobile user interfaces, *frApps* (short for *Frappé Apps*), which can be delivered directly through QR codes without requiring any network connectivity (Figure 1). Frappé achieves this by decomposing the UI into its essential metadata and corresponding API linkages and then encoding them within a QR code, while offloading the UI functionalities to API calls. We developed a Frappé reconstructor Android application, *reFrappé*, capable of reconstructing the UI from the metadata ad-hoc on the user's device when the user scans any QR code encoded with a frApp. Finally, we integrated our Frappé framework within a WYSIWYG UI builder, *enFrappé*, to streamline the process of creating frApps. Using enFrappé, developers can design frApps through simple drag-and-drop gestures and then deploy them in the form of QR codes.

We envision Frappé as a building block for augmenting the interactive capabilities of QR codes. While the user interfaces designed using our Frappé framework are not meant to be a replacement for full-featured native apps, we believe its quick and network-free deployment makes it ideal for ephemeral usage scenarios in areas with sparse network connectivity, such as guide apps for national parks and information systems at airports. Beyond that, Frappé can also serve as a quick gateway to enhance the way we interact with everyday objects by allowing easy integration of software with physical objects. For example, Frappé can be used to create lightweight controllers for smart devices, allowing users to control them directly without installing any device-specific application.

[1]Frappé = **F**ramework for **R**apid **AP**I-based **P**rototyping and **E**nvironmental Deployment

We conducted a technical evaluation of our Frappé framework where we measured the size, network usage, and load times of sample frApps against a benchmark of identical web and native applications. Our findings indicate that frApps exhibit a notable performance advantage when compared to its web and native app counterparts. Furthermore, we conducted a user study with 16 participants of various app designing backgrounds to evaluate the usability and utility of our framework. Both expert and novice participants found the framework easy to use and reported high learnability of the system. Participants also reported a high utility value of our framework and suggested several real-life use-case scenarios for Frappé.

We summarize our contribution as follows:

(1) We present Frappé, a framework for deploying ultra lightweight UIs directly to mobile devices through QR codes, without requiring any network connectivity.
(2) We also introduce enFrappé, a WYSIWYG UI builder for developing frApps.
(3) We perform a technical evaluation to measure the performance of frApps while keeping identical web and native app counterparts as benchmark. Furthermore, we evaluate the usability and utility of our framework in allowing developers to design, deploy, and use these lightweight functional mobile UIs.

2 RELATED WORK

Our research draws inspiration from the related works in three categories - (1) on-demand application delivery techniques, (2) interactive uses of visual markers, and (3) rapid application prototyping techniques.

2.1 On-Demand Application Delivery

Prior works from both industry and academia have explored on-demand delivery of mobile applications to allow users to quickly use various apps without downloading and installing them on their devices. Google's *Play Instant* [17] and Apple's *App Clips* [24] allow developers to create a smaller version of their application with limited features which can be loaded on the end-user's device almost instantly. These instant apps offer a rich UI and a native-app-like experience, but have limited access to device resources and cannot perform background activities [9, 18, 25, 36]. Moreover, they require significant effort on the developer's side to refactor the preexisting full-sized app into its smaller counterpart [9].

Full-scale native Android applications usually feature multiple functionalities. However, during a single use, an end-user may not need all the functionalities that the application provides. Based on this premise, Bhardwaj et al. [9] proposed the use of *app slices* to deliver lightweight mobile apps to end-users. They implemented *AppSlicer*, a script to automatically fragments full-scale native apps into several lightweight *slices* containing individual app functionality, without any development overhead. These slices can be dynamically served over a content delivery network (CDN) based on the user's needs, thus preventing the need for downloading the entire application. However, loading these app slices dynamically requires a fast connection to the CDN for an uninterrupted user experience.

Standard user-centric Internet of Things (IoT) devices require users to have device-specific mobile applications. This approach becomes unscalable as the number of such devices increase [47]. IoT ecosystems such as *Google Home* [16], *Apple Homekit* [23], *Alexa Smart Home* [1] and *Samsung SmartThings* [43] attempt to solve this problem by grouping controls for compatible devices under a single unified application. Thus, instead of having several individual apps for each device, a single app can dynamically load the control for all the *compatible* smart devices that are connected to the ecosystem. However, this integration is mostly limited to smart devices in home and office environments [47, 48].

The Web of Things (WoT) uses and extends pre-existing web standards defined by the World Wide Web Consortium (W3C) to promote interoperability and enable communication between *smart things* and web applications [19, 20, 49]. Furthermore, Progressive Web Apps (PWAs) allow for a rich, native-app-like experience and can run directly on web browsers without the download and installation overheads [12]. Zachariah et al. proposed an architecture for enabling BLE-enabled smart devices to advertise their controls through web apps [47]. They implemented *Summon*, an Android app that scans for available BLE-enabled smart devices and integrates their web apps within the native application environment, allowing for both persistent and ephemeral use. Web apps have also been combined with augmented reality (AR) to provide a more intuitive control for smart devices [6, 48]. Such techniques involve localizing the smart devices in the environment, and then overlaying their corresponding web app over a floating *WebView* component inside the AR environment. Web apps are ideal for ephemeral use cases where the user does not need to download and install anything permanently, but they are limited by their ability to access device resources and background services. While service workers in PWAs solve this problem by allowing rich interactions such as background script execution and push notifications, they introduce new attack vectors which could allow for resource abuse [33]. Moreover, the iOS *WebKit* does not allow access to a majority of device hardware components, including the Web Bluetooth and NFC API [26], thus restricting the ubiquity of such approaches.

Cloud-based application virtualization services such as *Numecent Cloudpager* [40], *Microsoft Azure Virtual Desktop* [39], and *VMware Horizon* [46] allow users to offload desktop applications on the cloud and stream them directly to their devices. However, these services are expensive in terms of cost, network, and power usage, and they do not have access to on-device sensors and background services which significantly reduces their scope of use [9]. *AppFlux* [7] and *Ephemeral Apps* [8] improves the loading time for streaming native Android apps, allowing the end-users to try out the app before installing it.

All the approaches for on-demand application delivery discussed above are heavily reliant on internet connectivity to load the application. IoT ecosystems and app streaming services require the user's smartphone to be always online for proper operation, whereas instant apps and app slices are typically limited to a few MBs, which is further reduced in the case of web apps (~100KBs). While these solutions significantly reduce the network connectivity overheads while supporting a rich set of features, the network demands may still be too high for ephemeral usage scenarios and for areas with limited network connectivity, which can act as a barrier to app acceptance, or even worse, prevent access to essential services. Frappé can generate UIs weighing less than a kilobyte which can be encoded entirely into a QR code. Unlike the previous works, Frappé does not require an internet connection to load the application's UI, and will require internet *only if* it is needed by the app's functionality. Moreover, since an uncompressed web request header typically weighs around 700-800 bytes [15], Frappé's network usage, if any, will be significantly less than loading entire web apps.

2.2 Interactive Uses of Visual Markers

QR codes can encode a relatively high capacity of data (up to 2953 bytes with low error correction level), are cheap and easy to produce, and can be easily read by modern smartphones [27]. They are ubiquitously used today to serve as a portal to digital media [3]. Organizations advertise their websites by embedding their hyperlinks in QR codes. During the COVID-19 pandemic, several restaurants digitized their menus through QR codes to enable contactless menu browsing. Advancements in mobile deep linking allow the use of uniform resource identifiers (URIs) to link directly to specific parts of a mobile application. These deep links have been integrated within QR codes to enable mobile payment systems and to connect smart devices with their native apps.

Visual markers have also been used for more interactive purposes, such as enabling mobile devices to be used as input devices and storing and transmitting user interfaces. As the visual markers' shape seen by a camera reflects the relative position and orientation of the camera, researchers demonstrated the use of such markers to allow users to control virtual content on a digital display [4, 5] and physical paper [38] using a mobile device. The visual markers' high information capacity also enabled storing various types of user interface controls, such as menus, buttons, and sliders, so that the users can access and manipulate them on their mobile devices without using Bluetooth or mobile network [42].

Researchers and developers have also shown that the QR code can store a simple working program for the users to launch the program on their mobile device without using the internet to download it. [11, 37, 44] For example, a Microsoft Windows executable for the traditional snake game was encoded entirely into a QR code, allowing users to launch the game directly upon scanning it [37]. This approach was, however, exclusive to this instance and does not offer a generalized framework for creating and deploying other kinds of apps using a QR code. On the other hand, *Rewtro* [11] and *QRGame* [44] featured a generalized game development framework that can allow users to create and encode their own games entirely in a QR code for an internet-free deployment. These techniques are able to encode the games directly into the QR code due to the lightweight nature of the 8-bit-esque games, which are usually of a few KBs. They also showed that using multiple visual markers [11, 35] can further increase the data capacity for larger programs and media content.

Previous research has shown the potential of utilizing visual markers to store UI controls and applications, which can then be deployed by placing the markers in the environment. However, the challenge of creating applications optimized for QR code-based deployment remains, as it would require careful implementation from scratch by a skilled software developer. Our framework builds upon these existing works and offers an easy and efficient solution for designing, developing, and deploying functional mobile user interfaces without the need for a network connection.

2.3 Rapid Application Prototyping

It is critical for developers to be able to prototype and evaluate app designs quickly, and prior works have explored various techniques for such rapid prototyping of design ideas. For instance, *Mallard* [50] allows users to test and prototype pre-trained machine learning models directly on a browser by manipulating and scaffolding readily-available data from webpages. *Stylette* [30] is another system that accepts natural language speech commands to interpret the intent of developers and suggest a palette of relevant CSS properties and values. Programming by demonstration (PBD) has been extensively explored to allow developers to rapidly develop functional prototypes [10, 21, 22, 29, 34]. *Sketch-n-sketch* [22] explores a variety of output-directed techniques to allow dynamic code generation for scalable vector graphics (SVGs) through direct object manipulation. *Umitation* [10] allows web developers to mimic UI behaviours from pre-existing websites. *SUGILITE* [34] accepts natural language speech commands from users to automate arbitrary tasks in any Android app. d.mix [21] is a tool for creating website mashups by enabling users to browse annotated websites and select elements for d.mix to sample. Mobile application development can also benefit from rapid application prototyping. On a more general level, *X-Droid* [29] allows developers to *borrow* functionality from other pre-existing Android applications, significantly reducing development effort. Lastly, there are several low-code/no-code online tools that allow users to build mobile apps by simply dragging and dropping the UI controls from a toolbox to the app screen [2, 41, 45]. Our framework draws inspiration from such methodologies to facilitate rapid application prototyping with the consideration for generating ultra-lightweight applications that require minimal network connectivity to function.

3 FRAMEWORK DESIGN PRINCIPLES

Following our literature review, we based the design of our framework on the following principles:

Scope. Frappé is not meant to act as a replacement for the rich functionalities offered by traditional native apps, but instead serve as an extremely lightweight access point to access essential services dynamically without the barriers of extensive internet bandwidth consumption, memory usage, or performance penalties.

Minimal Internet Dependence. Frappé should not require internet to load the user interface. Internet connectivity should only be used for the purpose of functionality, *if necessary*, to make API calls. Furthermore, if the API call is done over LAN, BLE, or NFC, internet bandwidth should not be used at all, making the user interface completely independent of the internet connectivity.

Minimal Usage Barrier. Frappé should allow users to instantly access the services provided by its apps without the overhead of downloading and/or installing the app. Frappé-compatible apps should not consume any significant amount of disk space or require any specific hardware beyond a standard smartphone.

Optimized for Transient Use. Frappé-compatible apps should not clutter the user's app drawer. If an app is designed for a one-time use case scenario, e.g., app for registering entry to an event, it should be removed as soon as it serves its purpose. If an app is designed for a transient use case based on the user's environmental context, e.g., an information app for a national park, such apps should be directly available to the user if the user is present within that environmental context, whereas if the user is away, such apps should hide itself from direct view but can still be accessed if the user specifically searches for it. The same idea can also be extended to repeated transient use based on the user's environmental context. For example, an app for controlling a specific smart light bulb may be visible to the user when the user is present near the smart light, and remain hidden if the user is away.

Minimized Development and Deployment Overheads. Adoption of the Frappé framework should not introduce additional development and deployment overhead. Developing Frappé-compatible apps should be easy and should not require the developer to refactor preexisting code. Instead, Frappé should support the use of preexisting services through API calls. The deployment of Frappé-compatible apps should be easy and should not introduce additional hardware dependencies such as BLE beacons or NFC sensors.

4 FRAPPÉ

We designed Frappé to augment the interactive capabilities of QR codes by allowing them to serve ultra lightweight mobile UIs, frApps, directly to the end-user's device, without the need for any network connectivity. Our framework can significantly minimize the size of the designed UIs by reducing its user interface elements to a set of fundamental metadata, while offloading the functionality to external API calls. For example, a Button component can be deconstructed into a set of discrete properties such as *label* and *onPressEvent*. While optional, properties such as *textColor* and *backgroundColor* can also be included to enable some degree of customization. Conversely, our framework can reconstruct the UI from these reduced metadata, ad-hoc. While this concept of reconstructing elements from metadata is common in many cross-platform application development frameworks such as *Xamarin* and *React Native*, UIs created using Frappé are extremely lightweight, typically in the scale of a few hundred bytes. This lightweight nature allows us to encode and deploy frApps directly through QR codes.

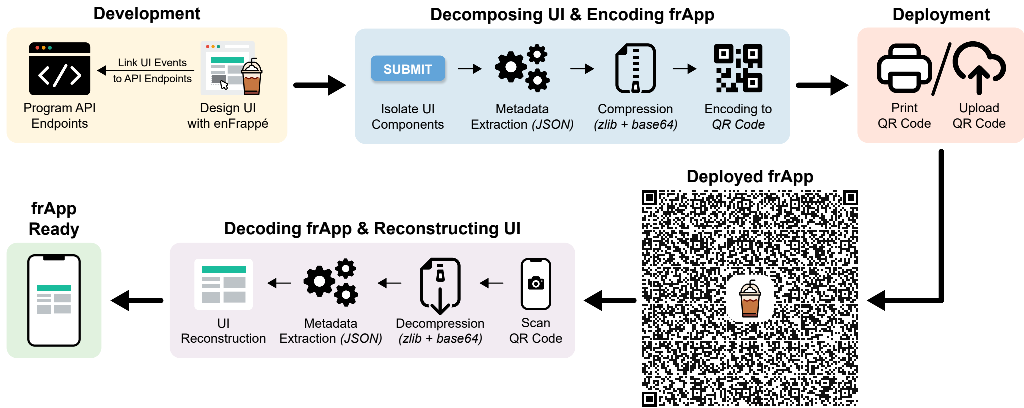


Fig. 2. System workflow for designing, deploying, and executing frApps using the Frappé framework

The highlight of our framework is its ability to execute UI functionality while requiring minimal network bandwidth. Frappé achieves this by leveraging the ultra lightweight nature of RESTful API calls, which typically only cost a few hundred bytes to be triggered [15]. Thus, by entirely offloading functionality to API calls and reducing the UI to its bare essentials, Frappé is able to serve the UI free of network-connectivity, while costing the minimum network bandwidth *only* for the functionalities required by the user. Finally, Frappé can support a completely network-free experience by queuing requested API calls in case the user is in an area with limited network connectivity, and scheduling it when the user has better network coverage, making our framework ideal for use in areas with sparse network coverage.

4.1 System Design and Implementation

We realize our framework through two components - (i) enFrappé, a tool for developers to design frApps, and (ii) reFrappé, a system for the end-users to access the designed Frappé apps. Our system has been made open-source on GitHub[2]. Figure 2 illustrates the workflow of our framework.

4.1.1 enFrappé – UI Designer Tool. We created a WYSIWYG UI builder tool, enFrappé, for streamlining the process of designing UIs using our Frappé framework (Figure 3). enFrappé's user interface resembles traditional IDEs for building mobile apps, such as *Android Studio* and *Xcode*. enFrappé has a *toolbar* for displaying all available UI components (Figure 3a), a panel for displaying the *properties* of any selected UI component (Figure 3b), a *prototyping area* (Figure 3c), and a panel for configuring *application settings* (Figure 3d). However, unlike traditional IDEs, enFrappé does not provide any environment for writing code. Instead, it features a simple drag-and-drop interface for arranging the UI components and uses API links to connect the UI events to the functionalities offloaded on a server (Figures 4a and 4b). While the API server can be manually written by the developer, enFrappé also supports automatic generation of a custom *Flask* server capable of receiving and displaying inputs from all API endpoints defined in the designed app (Figure 4c). This design choice allows enFrappé to be beginner-friendly while also providing expert developers with the ability to integrate custom functionality into their apps. enFrappé allows the developer to encode the designed user interface into a QR code. Given the limited storage capacity of QR codes, enFrappé

[2]enFrappé: https://github.com/adildsw/enfrappe
reFrappé: https://github.com/adildsw/refrappe

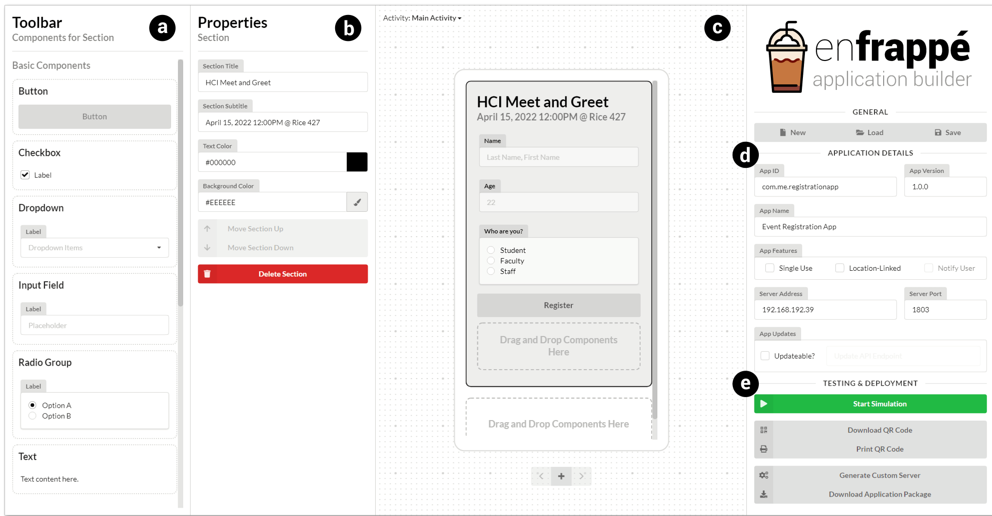


Fig. 3. enFrappé, a WYSIWYG UI builder for designing ultra lightweight mobile UIs using the Frappé framework. The interface can be divided into five sections: (a) toolbar panel containing available UI components, (b) properties panel containing properties of the selected component, (c) prototyping area for designing and visualizing the app, (d) application details panel containing app metadata, and (e) testing and deployment panel for running simulations, and generating QR codes and custom server.

first compresses the designed UI metadata using zlib compression [13], and then performs a Base64 encoding on it before storing it in the QR code. For complex UIs with several components, the size of the compressed UI metadata may exceed the QR code data capacity. In such cases, enFrappé splits the metadata into multiple QR codes.

4.1.2 reFrappé – Frappé Reconstructor Application. We developed an Android application, reFrappé, for reconstructing the UI of the frApp from the QR code. To launch a frApp, the end-user can simply use the camera app on their phone, or use the QR code scanner provided within reFrappé to scan the QR code. Upon successful scanning, reFrappé reconstructs the user interface of the frApp on the user's device. reFrappé also stores the metadata of all the previously scanned frApps, and users can easily access them using a frApp list, without having to rescan the QR code. When the user selects a frApp from the list, it gets reconstructed immediately on an ad-hoc basis, and once the frApp is no longer in use, it gets deconstructed, thereby releasing the consumed device resources. Figure 4b illustrates a reconstructed frApp designed in enFrappé (Figure 3) running on the user's device.

4.2 Designing and Deploying frApps

In this section, we demonstrate the workflow of designing and deploying frApps using enFrappé.

4.2.1 Designing User Interface. When the developer launches enFrappé, the prototyping area is initialized with a blank white activity (page) called *Main Activity*. Since no component is selected at the beginning, the user interface shows an empty *toolbar* and *properties* panel. The developer can populate the toolbar and properties panels with relevant items upon selecting the component of interest in the prototyping area. Clicking on the *Main Activity* populates the properties panel with Activity properties such as *Activity Name* and *Background Color*. However, the developer still does

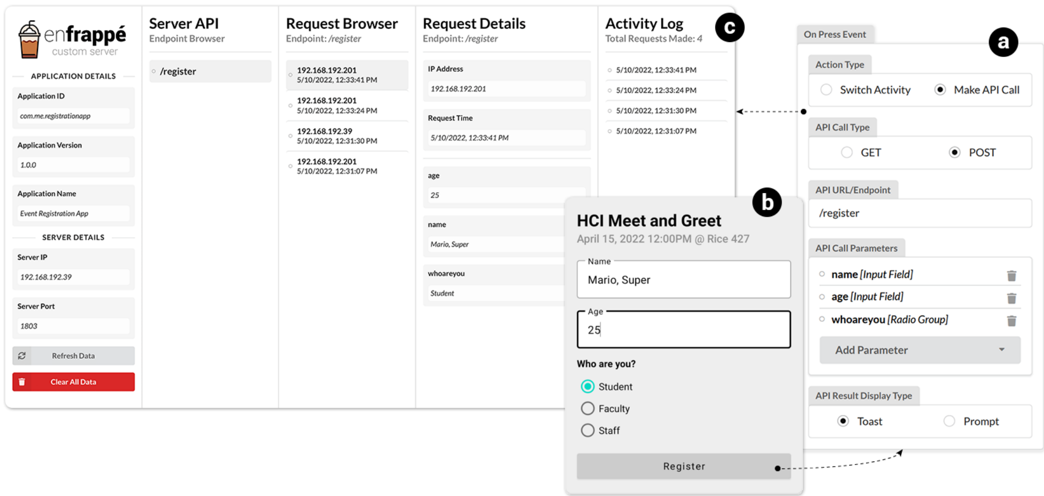


Fig. 4. Linking UI Components with API Calls. (a) The On Press Event is defined to make an API call with the required parameters; (b) the register button in the UI is linked to the defined On Press Event; (c) custom server generated by enFrappé receives and visualizes the API call requests made to the server.

not have access to basic UI elements such as Buttons, Checkboxes, and Radio Buttons. The Activity component supports only *Section* components to be placed within it, which serve as containers for all the other UI elements. Thus, before the developer can begin adding basic elements to their UI, they need to drag and drop a section component from the toolbar panel into the activity. The developers now have access to all the other UI elements which they can add to the section. All the UI components that are currently supported by enFrappé, and the properties panels for some of the UI components are illustrated in the appendix. enFrappé only allows components within the same section to be stacked vertically. The developer can add multiple sections to an activity. The developer also has the option of adding additional activities by pressing the "+" button at the bottom of the prototyping area. However, the *Main Activity* serves as the starting point of the UI and will be the first thing that appears when the UI is launched on the end-user's device. Using these features of enFrappé, the developer can drag and drop components and modify their properties to design their required user interface.

4.2.2 Connecting User Interface to Functionality. enFrappé integrates functionality with the user interface by linking API calls to UI events. The current build of enFrappé only supports the *OnPress* event for the *Button* component, and the *OnLoad* event for the *Chart* and *DataViewer* components. To make an API call, three parameters must be defined: *API Endpoint/URL*, *API Call Type* (GET/POST), and *API Call Parameters* (data that needs to be sent along with the API call) (Figure 4a). enFrappé binds component data values with API parameters through a property called *Parameter* Name. Every UI component that can take user input such as *Input Fields*, *Dropdowns*, and *Radio Buttons* has the *Parameter Name* property, which is used as an identifier for referring to their stored values. To add a parameter to an API call, the developer can simply click on the *API Call Parameter* property, which reveals the Parameter Names of all the components that exist in the designed user interface. The developer can then select the parameter values that they want to send along with the API call. enFrappé also allows the end-users to see the response of the API calls through either a *Toast*, or

a *Prompt*. In addition to making API calls, the Button's *OnPress* event can also be used to switch between activities.

While developers can code their own API servers for Frappé apps, enFrappé also allows users to generate their custom servers without writing any code. To generate a custom server, the developer can first populate the event properties of all relevant components in the interface with the required *API Endpoints*, *Parameters*, and *Call Types*. Then, the developer can simply click the *Generate Custom Server* button under the *Testing & Deployment* section (Figure 3e). enFrappé generates a custom *Flask* back-end server designed to receive and store all requests made to the APIs defined above, and a front-end web app to visualize API requests. A sample *OnPress* API configuration and the generated custom server interface are illustrated in Figure 4.

4.2.3 Deploying Application. In the *Application Details* panel (Figure 3d), the developer needs to provide the *App ID*, *App Name*, and *App Version* for the application. If their application makes API calls, they also need to provide the *Server Address* and *Server Port*. enFrappé also provides the option of specifying whether the frApp is intended to be a *Single Use* or *Location-Linked* to facilitate transient app usage. If a frApp is meant to be used only once, such as a registration app, then the end-user has no requirement for it once its purpose has been served. Such frApps can be marked as *Single Use*, and doing so will not store the frApp metadata on the end-user's device. Similarly, *Location-Linked* frApps are expected to be used only when the end-user is in the proximity of a certain location, such as controlling the smart light of the bedroom. If a frApp is marked as *Location-Linked*, that frApp does not appear in the frApp list until the user is in the required proximity. Furthermore, *Location-Linked* frApps can also be configured to *Notify User* if the frApp is available for use.

enFrappé allows the developer to test their frApp directly in the enFrappé environment by clicking the *Start Simulation* button under the *Testing & Deployment* section. Once the developer is satisfied with their frApp, they can click on the *Download/Print QR Code* button to generate a QR code of their frApp's compressed UI metadata. The frApp can be deployed by releasing a copy of the generated QR code to the end-users. This can be accomplished on various scales - for local deployment, printed copies of QR codes can be pasted on accessible surfaces such as walls and notice boards; alternatively, for a larger outreach, the QR code can also be hosted on any online platforms such as personal websites.

5 APPLICATION SCENARIOS

While Frappé is not designed for developing applications with rich user interfaces, there are several scenarios that can benefit from Frappé's lightweight nature and rapid deployment technique. We group our proposed application scenarios under three categories: (i) data collection, (ii) interfacing with physical objects, and (iii) controlling virtual systems.

5.1 Data Collection

If a user wants to sign up for a service or register for an event, they are typically required to access online form services such as Google Forms. However, limited network connectivity can pose a challenge for the user to access these services. Form apps typically require only a basic set of UI components (such as input fields, checkboxes, and radio buttons) to accept user input and pass it to the server, making it one of the most obvious use cases for Frappé. Using enFrappé, anyone can easily design a form app and generate a server capable of receiving and storing the input without writing a single line of code. Moreover, developers can easily integrate the output generated by the form app with their custom code, which usually requires several steps for typical online form services. Frappé is ideal for this use case as the user can access the form instantly

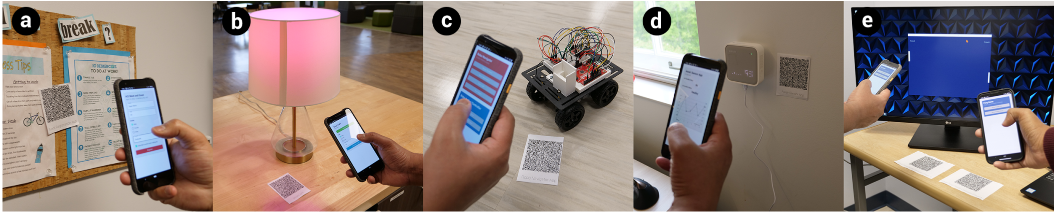


Fig. 5. Some application scenarios for UIs designed using Frappé. (a) Registering for an event; (b) Controlling a smart light; (c) Controlling a robot; (d) Reading sensor values; (e) Playing a 2-player Pong game.

by simply scanning the QR code with their device, and minimal network bandwidth is required to submit their response to the server. Furthermore, in case of no network connectivity, Frappé can queue the response submission for whenever the user reaches an area with better network coverage. Figure 5a illustrates an event registration app made using Frappé.

5.2 Interfacing the Physical Objects

Frappé's network-free rapid deployment technique can potentially allow seamless interfacing with physical objects. Smart home devices, such as smart light bulbs, usually require a device-specific app to control them. Moreover, different brands of smart devices may require installing separate apps, leading to increased network bandwidth consumption. This can be particularly challenging for users in unfamiliar environments, who may be locked out of the smart device features for not having the associated apps pre-installed on their smartphones. Frappé can be used to create lightweight UIs that can control these smart devices (Figure 5b). Since Frappé can serve UIs instantly, it can find significant applicability in making museums and exhibits more interactive by allowing users to interact with demonstrations seamlessly through frApps. Frappé can also be extended to create UIs that can interface with IoT sensors and actuators. Consider a temperature sensor – by simply scanning a QR code, a user can visualize the temperature trends over the last week through a chart (Figure 5d). Controls for physical devices such as toys and robots can also be offloaded to buttons on frApps, providing immediate access to the control to users without downloading device-specific controller apps (Figure 5c).

5.3 Controlling Virtual Systems

Since Frappé connects the UI with functionality through API calls, frApps can be used to trigger remote actions. For instance, Frappé can be used to make a gamepad for a computer game, or as a media controller for controlling music playback on remote speakers. In essence, UIs made using Frappé can be linked to any pre-existing system's code through API calls, allowing users to remotely trigger such functionalities. Figure 5e shows two players playing the Pong game using Frappé apps.

6 TECHNICAL EVALUATION

We performed a series of technical evaluations to measure the network, memory, and performance overheads of using Frappé. All technical evaluations were performed using 5 sample frApps: (i) basic hello world app, (ii) temperature sensor value reader, (iii) virtual pong paddle controller, (iv) smart light controller, and (v) event registration app. To benchmark the load times and network usage of frApps against similar web and native apps, we created identical versions of these five frApps using ReactJS and Android Studio. The size of the Android app bundle for the Frappé reconstructor

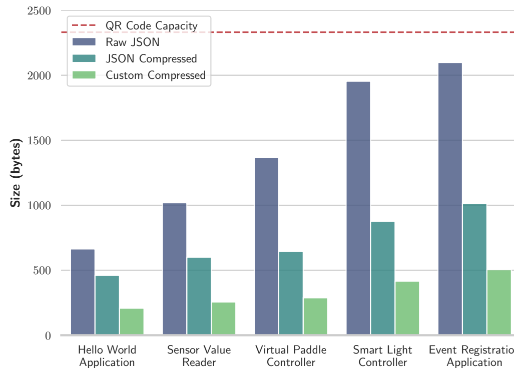


Fig. 6. Compressed and uncompressed UI metadata sizes of 5 sample frApps

app is 4.9 MB, and its load time was recorded as 783 ms on a Google Pixel 3a smartphone running Android 11. All subsequent tests were conducted on the same device.

6.1 Frappé App Size Analysis

Figure 6 illustrates the raw and compressed sizes of the UI metadata required to reconstruct the frApps on the user's device. The compressed data size of all our sample frApps remain below the maximum capacity of a QR code with medium level error correction capability (2331 bytes). From our results, we can observe that as the raw JSON data size increases, there is also an increase in the compression ratio. For large raw JSON data sizes, we achieve a compression ratio of over 2:1 by performing a simple Base64 encoding over a zlib compression. In our implementation, we used JSON for its convenience. However, considering the redundancy of the JSON data format, switching to a custom data format can help us further reduce the data size. For example, eliminating all redundancies (e.g., removing keys and using enumerations) in the event registration app data results in a compressed data size of only 504 bytes (24.0%) from its original size of 2098 bytes, whereas our currently employed compression technique only reduces the size to 1012 bytes (48.2%).

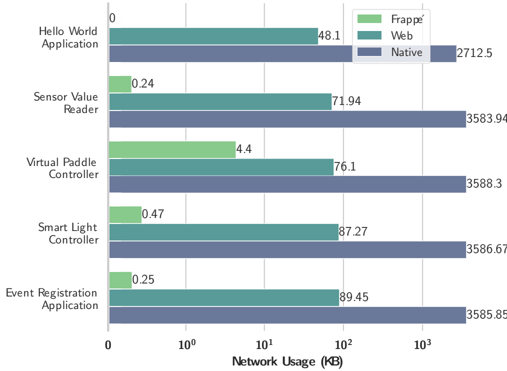
6.2 Network Usage

Figure 7a illustrates the network usage of the aforementioned apps across the three platforms. In our measurements, we consider the network usage incurred in obtaining the app on the device and using its functionalities. This means, for web apps we measured the bandwidth spent on loading the web app, whereas for native apps we measured the bandwidth spent on downloading the native app. Since the frApps are deployed as QR codes, no network bandwidth usage is required for obtaining frApps on the device. The network bandwidth usage for API calls is common to all three platforms, which is elaborated further in Table 1.

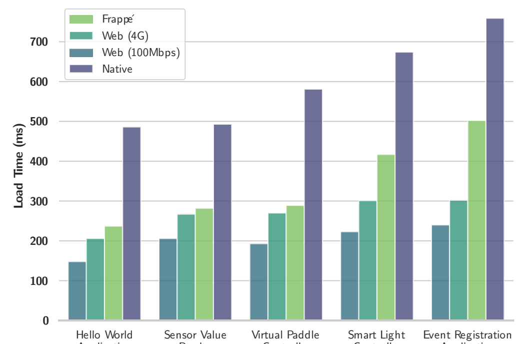
Our results demonstrate that the network bandwidth consumed by frApps is less than that of web apps and native apps by multiple orders of magnitude. While the other platforms consume significant amounts of network bandwidth for structural purposes, e.g., loading the user interface, frApps only consume network bandwidth for functional purposes, i.e., making API calls. It is worth noting that downloading the native apps is a one-time cost, and every successive use of the app will result in bandwidth usage similar to that of Frappé apps. Web apps can also be cached, thus reducing the total recurring network bandwidth usage. However, the low initial network cost of Frappé apps can encourage app acceptance, especially for single-use ephemeral apps.

Table 1. Analysis of the various API calls made during our technical evaluation

Application Name	Total API Call Network Usage	API Call Description
Hello World App	0 bytes	No API calls were made.
Sensor Value Reader	248 bytes	1 API call was made for obtaining sensor value.
Virtual Paddle Controller	4500 bytes	20 API calls were made for performing 20 paddle movement commands across a 3-life Pong game.
Smart Light Controller	479 bytes	2 API calls were made, one for switching on the smart light, and another for customizing the color and brightness of the smart light.
Event Registration App	252 bytes	1 API call was made for registering user by sending user's name, age, gender, pursuing degree, and agreement to terms and conditions.



(a) Network Usage for each application. (A logarithmic scale was used for plotting the network usage results)



(b) Load times for each application

Fig. 7. Comparison of performance overheads between 5 sample frApps and similar web and native apps

6.3 App Load Time

Figure 7b illustrates the load times of the aforementioned apps for the three platforms. For measuring the precise impact of each app on the load time, we did not include the load time of the web browser (for web apps) and the Frappé reconstructor app (for Frappé apps) in our readings. To get a better perspective of load times in different environmental settings, we considered two different network speeds for web applications - (i) 100Mbps for an indoor WiFi setting and (ii) 5.1Mbps for an outdoor setting with typical 4G connectivity [14]. Additionally, for both the web app network settings, we set the network latency to 0ms.

Our results demonstrate that the loading time of Frappé apps are comparable to those of web apps and native apps. We also considered situations such as traveling to remote areas or national parks where one might only get a 3G or 2G connection. For network speeds of 1Mbps and 100kbps, even the basic Hello World web app takes well over 1s and 4s to load, respectively.

7 USER STUDY

We conducted a user study to gain initial insights into the enFrappé's usability and utility in designing functional mobile user interfaces using the Frappé framework. The design of this study was inspired by the system evaluation of *Umitation* [10] and the usage evaluation from the *Evaluation Strategies for HCI Toolkit Research* [32]. In this study, we recruited participants with varying programming and application design experiences. The purpose of this was threefold: we wanted to (1) qualitatively measure the learnability of enFrappé by observing novice UI designers, (2) observe how expert UI designers interact with enFrappé, and (3) seek feedback on the utility value of Frappé based on their diverse range of experiences in designing UIs and using mobile apps available on the market. The study was approved by the institutional review board.

7.1 Participants

We recruited 16 participants (5 female, 11 male) from our university and nearby localities through email groups and word of mouth, with participants' ages ranging between 20 and 30 years (M=26.63, SD=2.63). Based on the participants' experience in UI design and programming, we categorized them into three groups - novice, intermediate, and expert. Participants who had several years of experience in coding *and* UI design were categorized as experts, whereas participants with little to no coding *or* UI design experience were categorized as novice. Participants who had some coding *or* UI designing experience were categorized as intermediate. Following our categorization, we had 6 novice participants (P2, P4, P5, P10, P13, P16), 4 intermediate participants (P6, P11, P12, P14), and 6 expert participants (P1, P3, P7, P8, P9, P15). Among the six novice participants, three (P2, P13, P16) had absolutely no experience in coding or UI designing, with two of them (P2, P13) possessing only rudimentary knowledge of using a computer. Three of our six expert participants (P3, P7, P9) had prior experience in developing apps in the industry, and one expert participant (P15) was extensively involved in organizing hackathons for the local university. The participants were compensated with 25 USD for their time.

7.2 Study Design

Each user study session lasted for approximately 50 to 60 minutes. The experiments were conducted in-person in a lab using the experimenter's computer and smartphone. At the beginning of each session, the participants' demographic details were recorded and a short pre-task interview was conducted to gauge their coding and UI design experience. The participants were also asked about their familiarity with basic UI components (buttons, checkboxes, etc.) and RESTful APIs. Based on their response, the experimenter guided the participants through a tutorial of enFrappé that was tailored to their level of understanding.

Following the tutorial, the participants were asked to design three frApps. They were provided with a design guideline for each task containing details about the UI requirements and the associated API endpoints. The tasks required them to assume the role of a front-end developer, so they were only required to design the user interface and link them to the provided API endpoints. The participants were not required to code these API functions - they were either autogenerated by enFrappé, or pre-coded by the experimenter. Each task was considered complete when the participants were able to execute the UI on the mobile device with the desired functionality.

The participants were allowed to freely explore enFrappé and personalize the apps as per their choice. We recorded the time taken by the participants to complete each task. However, the participants were not aware of being timed and completed the tasks at their own pace. The participants were also allowed to request for assistance as and when required. After the completion of all the tasks, we conducted a semi-structured interview with the participants to learn about

their experiences with enFrappé. Finally, the participants were debriefed about the features of our Frappé framework, and were then encouraged to ideate their own application scenarios for it.

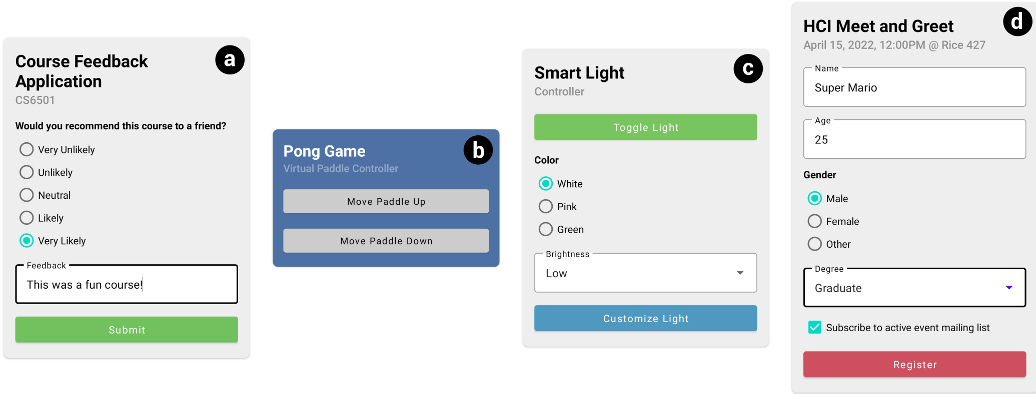


Fig. 8. UI screenshots of the 4 tasks that were performed during the user study. (a) Course Feedback App (Tutorial); (b) Virtual Pong Paddle Controller (Task 1); (c) Smart Light Controller (Task 2); (d) Event Registration App (Task 3)

Table 2. Description of all the tasks performed during the user study

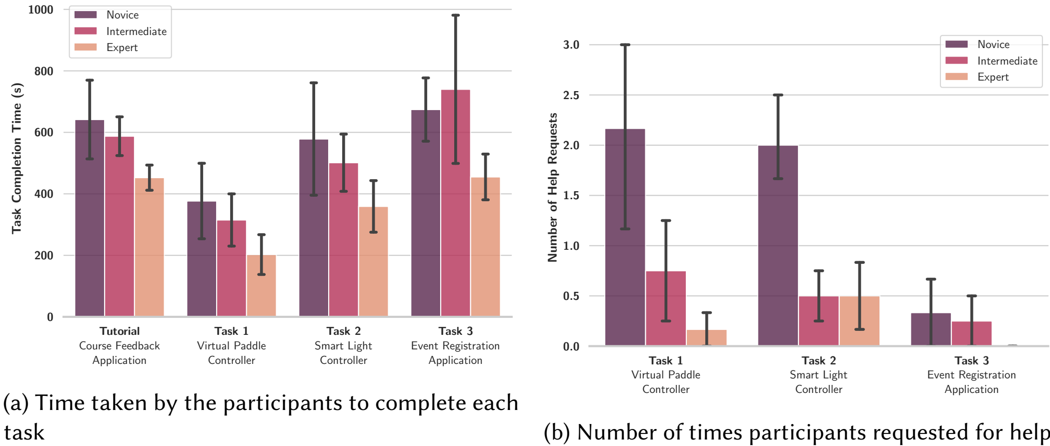
Task #	Application Name	UI Elements Involved	API Server Provided?	Task Description
0	Course Feedback App	Radio Button Input Field Button	No	*Tutorial task* performed by the experimenter to demonstrate how to design and deploy an application, and how to generate a backend server using enFrappé.
1	Virtual Pong Paddle Controller	Button (x2)	Yes	This task required minimal UI elements and was used as the first task to allow participants to get acquainted with the basic functionalities of enFrappé.
2	Smart Light Controller	Radio Button Dropdown Button (x2)	Yes	This task required participants to make parameterized API calls by passing the values of dropdown and radio button to the button's on-press event API call.
3	Event Registration App	Input Field (x2) Dropdown Radio Button Checkbox Button	No	The task required participants to incorporate all the basic UI components in their app, and create their own backend server using enFrappé to receive the inputs made in the designed app.

7.3 Tasks

The participants were introduced to enFrappé with a tutorial task that involved making a *Course Feedback App*. After the tutorial, the participants were given three tasks - *Virtual Pong Paddle Controller*, *Smart Light Controller*, and *Event Registration App*. Since we had participants of varying programming and UI design expertise levels, we designed our tasks in an increasing order of difficulty, with the first task serving as an entry point to the enFrappé system, and the last task exploring all the features that enFrappé has to offer. The user interfaces for all tasks are illustrated in Figure 8, and the task details are summarized in Table 2.

7.4 Results

Here we present the quantitative and qualitative analysis of our user study. First, we discuss the performance of the users in completing the tasks in terms of *time and accuracy*. Second, we discuss the perception of users towards our framework based on the feedback we received from the interview. For this, we conducted an inductive thematic analysis on all comments to identify the main themes. We then grouped the comments into three themes: (1) *ease of use*, (2) *learnability*, and (3) *utility*.



(a) Time taken by the participants to complete each task

(b) Number of times participants requested for help

Fig. 9. User study results

7.4.1 Time and Accuracy. All the participants were able to successfully complete all three tasks. We illustrate the time taken for the participants to complete each task, and the number of times assistance in Figure 9a and Figure 9b, respectively. Our results conform to the expectation that participants with more programming and app design experience could complete the tasks faster while requiring fewer interventions. The only outlier to this trend was participant P6 who explored our tool beyond the required design specifications and thus required more time to complete the third task.

The mistakes made during task completion were primarily because participants forgot specific elements of the instructions. *“Anything I messed up with was just me forgetting to do it. It wasn’t anything user interface-wise.”* (P16). Another common issue among novice and intermediate participants was that some of them were not familiar with certain technical terms such as API and component parameter names. *“I was just trying to understand the mechanism behind the API calls”* (P6). However, the participants became familiar with the system by the third task, as is evident from the decline in the number of help requests observed in Figure 9b.

7.4.2 Ease of Use. After completing the tasks, all participants reported that developing Frappé apps was easy. The participants described their experience of designing apps using our tool as *“straightforward”* (P6, P7, P9, P11), *“simplistic”* (P1, P8, P15), *“seamless”* (P12), and *“user-friendly”* (P15). One participant (P16) appreciated the visual nature of our tool and how *“you can see what you add”*. Participant P6 had a similar comment about the user interface - *“You don’t have too many options, so you can narrow down your manipulation on the design process”*.

Three expert participants (P1, P8, P15) compared their experience of designing apps using our tool with that of using traditional app building tools - *“I really liked the fact that it worked. I have*

had many experiences where things didn't work, and it took forever to debug. This was very simple to use." (P1); *"If I were to be actually coding this, I feel like that would be more difficult than doing this"* (P15); *"In traditional Android Studio, you can move components by one-third or one-half, but Frappé gets the job done if you're not developing a huge app"* (P8). Participant P8 further commented *"Testing an app on Android Studio is so annoying - you have to connect it, download it and then install it. Then the Android phone freaks out because they're like, this is untrusted. So just scanning the QR code and having the app working there, it's great! And I guess it's just easy for everyone to create their own apps - they don't have to have any backend or frontend experience. If you want to send out a survey, you can just print a QR code and smack it on the wall, and you're taking a survey!"*

7.4.3 Learnability. Despite some mistakes made while performing the initial tasks, most participants reported that as they progressed through the tasks, they became more comfortable with the system. Some participants mentioned being mindful of their previous mistakes while working on subsequent tasks - *"After a few tries, I figured out what you should put as parameters and what should be used as values for different radio groups"* (P4). A novice participant (P2) mentioned being *"nervous towards the beginning, but as I kept progressing through the tasks, I got more confident"*. Some participants credited the tutorial for easing them into the system - *"I think because you gave me a demo on how everything worked, it was very easy for me to start off"* (P1). To further improve the learning curve, some participants suggested using a *"tooltip for providing suggestions"* (P3), *"question mark buttons for providing additional information"* (P6) or a *"hint box for explaining the technical terms"* (P12).

7.4.4 Utility. The participants came up with several interesting suggestions for potential use cases of Frappé pertaining to their field of interest. Several participants suggested using Frappé for form-based applications such as questionnaires (P5), registration apps (P1, P9, P15, P16) and apps to collect responses for in-class quizzes (P6). They argued that Frappé apps did not *"require internet connectivity"* (P9) to load and worked without the requirement of *"signing in"* (P1, P16) to an existing account to submit a response, and thus offered a clear advantage over traditional online forms such as Google Forms. *"If we just had a QR code like this, people could scan and submit it - it seems like that could be super helpful because Google Forms is similar, but it's just a lot more work"* (P16). Moreover, two participants pointed out that Frappé offered more *"customizability than Google Forms"* (P6, P15) and could even augment the forms with functionalities that go beyond simply "collecting the information" (P6), such as *"grading the answer"* (P6) in quizzes.

Other miscellaneous use case suggestions included a diagnostic tool for remotely located IoT devices - *"you just obtain the configuration data like a chart on your device and try figuring out the issue"* (P7) - and apps for advertising - *"changing advertisements in shopping centers by scanning the QR code and changing their advertisement on the IoT devices"* (P3). Participant P11 suggested designing a lightweight UI to control robots in his lab instead of purchasing *"a bunch of joysticks"* since *"they are not cheap"* and while *"not everyone has a joystick, everyone has a phone"*. Participant P3 mentioned their parents' struggle with installing apps and recommended frApps as a potential solution for users who are not technologically proficient - *"they can just know that the app is in the QR code. They can use the app simply by scanning the QR code"* (P3).

Lastly, participant P3 complimented enFrappé's feature to generate a custom server. *"I think this is a pretty solid framework. For most of these template-based app-builders, it's just the UI part. But till now I haven't found any application which can generate the server along with the application. I think that this is actually the most interesting thing about this tool."*

8 DISCUSSION

Our technical evaluation demonstrates the advantages of using Frappé apps over web and native apps in terms of app size, network usage, and load times. Participants in our study emphasized the ease of building and deploying applications using our framework and demonstrated high learnability of the tool, regardless of their prior experience with programming and app designing. In this section, we discuss some insights about our framework, present its limitations and explore options for improving future builds of Frappé.

8.1 Tradeoff Between UI Customizability and Framework Efficacy

8.1.1 Customizable Features. While Frappé allows some degree of UI customization, such as changing component colors and altering text sizes, it is fairly limited in its ability to design rich user interfaces. During the user study, participants asked about the possibility of incorporating several customizations in their app design - *"Is there any option to change the color for each radio option individually?"* (P5, P6); *"Do the buttons not go side by side?"* (P7); *"landscape mode"* (P11). One participant also asked about the possibility of incorporating advanced logic into the UI design - *"If I select one of the colors can I also change the color of the button to reflect that?"* (P7). However, there is an inherent tradeoff between customizability and app size. Since our primary aim was to serve the designed UIs directly through a QR code, and given the QR codes limited storage capacity, we need to limit the customization possibilities in our framework.

8.1.2 Beyond Form Apps. Our Frappé framework is not limited to designing form-like UIs with basic components such as input fields and buttons. The framework can be expanded to support more sophisticated UI components such as *Charts*, *Map Views*, and *Date Time Picker* as long as the UI component can be broken down into a set of discrete parameters which can then be used to reconstruct the component. As a proof-of-concept, we included three such components (*DataViewer*, *Chart*, and *Image*) in our current Frappé build and categorized them as *Miscellaneous Components* (illustrated in Appendix). However, every component comes with their own network bandwidth cost, e.g., the *Image* component will use a significant amount of network bandwidth to load an image. Thus, the tradeoff between each component and their impact on the overall network bandwidth needs to be considered before inclusion into a frApp.

8.1.3 Dynamic UI Rendering. One of the guiding principles behind Frappé is its minimal dependence on internet connectivity, i.e., no internet connection should be required for loading the UI. This results in frApp UIs being static by nature. However, bypassing this principle can allow Frappé to support dynamic UI rendering. When making an API call, the server can return activities (or pages) dynamically, akin to loading new pages on a website. Unlike web pages, however, the network bandwidth consumption for loading these dynamic activities is extremely low (~500 bytes) given the lightweight nature of our framework. This can allow for many interesting use cases such as designing intricate quick-access control systems and serving app content based on user inputs.

8.2 Limitations

8.2.1 Reliance on Reconstructor Application. The requirement of an additional reconstructor application on the end-user's mobile device to load frApps from QR codes can be perceived as a major limitation of our framework. However, we envision our work as a foundation for augmenting such interactive capabilities in QR codes. When QR codes were first introduced, devices needed a dedicated application for reading them, but with their increase in popularity, today's smartphones come with in-built QR code readers. Furthermore, several QR code protocols, such as those of WiFi configurations, are now baked into modern smartphones. We hope our work can act as a

building block for standardized protocols allowing QR codes to serve UIs, which can be built into smartphones in the future, eliminating the need for an additional reconstructor application.

8.2.2 Reliance on Internet Connectivity. While users do not need internet connectivity to load Frappé apps, the current build of Frappé relies on RESTful APIs for connecting the user interface with functionality. Thus, making API calls may require an internet connection if the API server is hosted over the web. To alleviate this, Frappé features the queuing of API call requests in case of no network coverage so that the request can be scheduled for when sufficient network coverage is available. Moreover, since API calls require a very small amount of network bandwidth [15] (Table 1), future iterations of Frappé can explore the use of other low-cost transmission mediums such as bluetooth low energy, near-field communication, infrared, and visible light communication, thereby completely obviating the need for internet connectivity throughout the interaction. We are also witnessing a trend where modern smartphones are equipped with satellite communication modules, which can be exploited for use in frApps deployed in risky areas for facilitating emergency communication. Lastly, the lack of reliance on internet connectivity can also yield privacy benefits. Since our framework can load apps (and potentially execute app functionalities) without any internet connectivity, this can prevent any third party from tracking the user's actions, which can be an advantage as compared to web and native applications.

8.2.3 Security Challenges. We acknowledge the security challenges that our framework presents. While visual markers such as QR codes are inexpensive and convenient, they are inherently prone to security vulnerabilities [31]. Since the frApps are deployed using QR codes, a malicious user can easily manipulate the QR code to reroute sensitive information to their own system [28]. This was also highlighted by two expert participants during our user study - *"If you have a QR code pasted, it becomes very easy for anybody to access it."* (P7); *"A malicious user can transfer and steal user data, so maybe the interpreter should scan the QR code before translating it into executable format"* (P3). In its current state of development, we do not recommend using Frappé apps to perform transactions of sensitive data. While it is beyond our current scope, it may be useful to investigate the use of digital signatures and encryption techniques, or switch to a more secure transmission medium to bolster users' trust in Frappé apps.

8.2.4 Exploring Alternative UI Transmission Media. Beyond the security challenges, visual markers can also have a negative aesthetic impact on surrounding environments, particularly if they are large, obtrusive, or not well-integrated into the overall design. On a broader context, the ultra lightweight nature of our framework can be leveraged to support other passive media such as RFID and NFC tags for the transmission of mobile UIs, offering greater flexibility and convenience for embedding UIs into physical objects and enabling seamless interaction between users and their environments.

9 CONCLUSION

In this paper, we introduce Frappé, a framework for deploying ultra lightweight mobile user interfaces, frApps, directly through QR codes. We achieve this by preserving only the essential metadata of the UI while offloading the UI functionalities to API calls. End-users can execute these frApps by simply scanning the QR code, and the frApp UI gets instantly reconstructed on the user's device, without requiring any network connection. We also designed a WYSIWYG UI builder, enFrappé, to streamline the process of designing and deploying UIs using our framework. Our findings from a technical evaluation reflect significantly less performance overheads for apps generated using our framework, as compared to similar web and native apps. We also conducted a user study with 16 participants from various UI designing backgrounds to evaluate our framework

in terms of its usability and utility. All participants found it easy to use our framework to complete the required tasks and reported high learnability and utility value, regardless of prior UI design experience. QR codes are ubiquitous, inexpensive, and have a low barrier of access, and yet their modern day use-cases are one-dimensional. We envision our work as building blocks for augmenting the interactive capabilities of QR codes, allowing them to not only serve as static redirection artifacts, but as dynamic tools for enhancing human-computer interactions.

REFERENCES

[1] Amazon. 2014. Understand Smart Home Skills | Alexa Skills Kit. https://developer.amazon.com/en-US/docs/alexa/smarthome/understand-the-smart-home-skill-api.html

[2] Appenate. 2011. What Makes Appenate Stand Out From The Rest? https://www.appenate.com/why-appenate/

[3] Seongbok Baik. 2012. Rethinking QR code: analog portal to digital world. *Multimedia Tools and Applications* 58, 2 (May 2012), 427–434. https://doi.org/10.1007/s11042-010-0686-9

[4] Rafael Ballagas, Michael Rohs, and Jennifer G. Sheridan. 2005. Mobile Phones as Pointing Devices. In *Pervasive Mobile Interaction Devices (PERMID 2005) - Mobile Devices as Pervasive User Interfaces and Interaction Devices - Workshop in conjunction with: The 3rd International Conference on Pervasive Computing (PERVASIVE 2005), May 11 2005, Munich, Germany*, Enrico Rukzio, Jonna Häkkilä, Mirjana Spasojevic, Jani Mäntyjärvi, and Nishkam Ravi (Eds.). LMU Munich, Munich, Germany, 27–30.

[5] Rafael Ballagas, Michael Rohs, and Jennifer G. Sheridan. 2005. Sweep and Point and Shoot: Phonecam-Based Interactions for Large Public Displays. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, OR, USA) *(CHI EA '05)*. Association for Computing Machinery, New York, NY, USA, 1200–1203. https://doi.org/10.1145/1056808.1056876

[6] Jagni Dasa Horta Bezerra and Cidcley Teixeira de Souza. 2019. smAR2t: a Models at Runtime Architecture to Interact with the Web Of Things using Augmented Reality. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES 2019)*. Association for Computing Machinery, New York, NY, USA, 124–129. https://doi.org/10.1145/3350768.3353818

[7] Ketan Bhardwaj, Pragya Agrawal, Ada Gavrilowska, Karsten Schwan, and Adam Allred. 2015. Appflux: Taming app delivery via streaming. In *Proceedings of the 2015 Conference on Timely Results in Operating Systems (TRIOS 15)*. USENIX Association, Monterey, CA, USA.

[8] Ketan Bhardwaj, Ada Gavrilovska, and Karsten Schwan. 2016. Ephemeral Apps. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications (HotMobile '16)*. Association for Computing Machinery, New York, NY, USA, 81–86. https://doi.org/10.1145/2873587.2873591

[9] Ketan Bhardwaj, Matt Saunders, Nikita Juneja, and Ada Gavrilovska. 2019. Serving Mobile Apps: A Slice at a Time. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3302424.3303989

[10] Yan Chen and Tovi Grossman. 2021. Umitation: Retargeting UI Behavior Examples for Website Design. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, New York, NY, USA, 922–935. https://doi.org/10.1145/3472749.3474796

[11] Francesco Cottone. 2022. Rewtro. https://github.com/kesiev/rewtro original-date: 2019-12-19T15:08:05Z.

[12] Giulia de Andrade Cardieri and Luciana Martinez Zaina. 2018. Analyzing User Experience in Mobile Web, Native and Progressive Web Applications: A User and HCI Specialist Perspectives. In *Proceedings of the 17th Brazilian Symposium on Human Factors in Computing Systems (IHC 2018)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3274192.3274201

[13] Peter Deutsch and Jean-Loup Gailly. 1996. *Zlib compressed data format specification version 3.3.* Technical Report. RFC 1950, May.

[14] Ian Fogg. 2021. Quantifying the speed bar for a reliable mobile experience. https://www.opensignal.com/2021/06/17/quantifying-the-speed-bar-for-a-reliable-mobile-experience

[15] Google. 2010. SPDY: An experimental protocol for a faster web. https://www.chromium.org/spdy/spdy-whitepaper/

[16] Google. 2016. Google Home. https://developers.google.com/home

[17] Google. 2017. Google Play Instant. https://developer.android.com/topic/google-play-instant

[18] Google. 2019. Create an instant-enabled app bundle. https://developer.android.com/topic/google-play-instant/getting-started/instant-enabled-app-bundle

[19] Dominique Guinard and Vlad Trifa. 2016. *Building the Web of Things: With examples in Node.js and Raspberry Pi* (1st ed.). Manning Publications Co., USA.

[20] Dominique Guinard, Vlad Trifa, and Erik Wilde. 2010. A resource oriented architecture for the Web of Things. In *2010 Internet of Things (IOT)*. IEEE, Tokyo, Japan, 1–8. https://doi.org/10.1109/IOT.2010.5678452

[21] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 2007. Programming by a sample: rapidly creating web applications with d.mix. In *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1294211.1294254

[22] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925

[23] Apple Inc. 2019. HomeKit. https://developer.apple.com/homekit/

[24] Apple Inc. 2020. App Clips. https://developer.apple.com/app-clips/

[25] Apple Inc. 2020. Choosing the Right Functionality for Your App Clip | Apple Developer Documentation. https://developer.apple.com/documentation/app_clips/choosing_the_right_functionality_for_your_app_clip

[26] Apple Inc. 2020. Tracking Prevention in WebKit. https://webkit.org/tracking-prevention/

[27] Denso Wave Inc. 2004. What is a QR Code? | QRcode.com | DENSO WAVE. https://www.qrcode.com/en/about/

[28] Pranjal Jain, Rama Adithya Varanasi, and Nicola Dell. 2021. "Who is Protecting Us? No One!" Vulnerabilities Experienced by Low-Income Indian Merchants Using Digital Payments. In *ACM SIGCAS Conference on Computing and Sustainable Societies* (Virtual Event, Australia) *(COMPASS '21)*. Association for Computing Machinery, New York, NY, USA, 261–274. https://doi.org/10.1145/3460112.3471961

[29] Donghwi Kim, Sooyoung Park, Jihoon Ko, Steven Y. Ko, and Sung-Ju Lee. 2019. X-Droid: A Quick and Easy Android Prototyping Framework with a Single-App Illusion. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 95–108. https://doi.org/10.1145/3332165.3347890

[30] Tae Soo Kim, DaEun Choi, Yoonseo Choi, and Juho Kim. 2022. Stylette: Styling the Web with Natural Language. In *CHI Conference on Human Factors in Computing Systems*. Number 5 in CHI '22. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3491102.3501931

[31] Katharina Krombholz, Peter Frühwirt, Peter Kieseberg, Ioannis Kapsalis, Markus Huber, and Edgar Weippl. 2014. QR Code Security: A Survey of Attacks and Challenges for Usable Security. In *Human Aspects of Information Security, Privacy, and Trust*, Theo Tryfonas and Ioannis Askoxylakis (Eds.). Springer International Publishing, Cham, 79–90.

[32] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3173574.3173610

[33] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 2018. Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1731–1746. https://doi.org/10.1145/3243734.3243867

[34] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6038–6049. https://doi.org/10.1145/3025453.3025483

[35] Xu Liu, David Doermann, and Huiping Li. 2008. A Camera-Based Mobile Data Channel: Capacity and Analysis. In *Proceedings of the 16th ACM International Conference on Multimedia* (Vancouver, British Columbia, Canada) *(MM '08)*. Association for Computing Machinery, New York, NY, USA, 359–368. https://doi.org/10.1145/1459359.1459408

[36] Yi Liu, Enze Xu, Yun Ma, and Xuanzhe Liu. 2019. A First Look at Instant Service Consumption with Quick Apps on Mobile Devices. In *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, Milan, Italy, 328–335. https://doi.org/10.1109/ICWS.2019.00061

[37] MattKC. 2020. Snake in a QR code. https://mattkc.com/etc/snakeqr/

[38] Beat Gfeller Michael Rohs. 2004. Using Camera-Equipped Mobile Phones for Interacting with Real-World Objects, In Advances in pervasive computing, Alois Ferscha and et al. (Eds.). *Books@ocg.at* 176, 265–271. Pervasive 2004; Conference Date: April 18-23, 2004.

[39] Microsoft. 2008. Azure Virtual Desktop | Microsoft Azure. https://azure.microsoft.com/en-us/services/virtual-desktop/

[40] Numecent. 2012. Cloudpager - Container Management for Windows Desktops. https://www.numecent.com/cloudpager/

[41] Appy Pie. 2010. Free Online Mobile App Builder. https://www.appypie.com/app-builder/appmaker

[42] M. Rohs. 2005. Visual code widgets for marker-based interaction. In *25th IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, Columbus, OH, USA, 506–513. https://doi.org/10.1109/ICDCSW.2005.140

[43] Samsung. 2012. SmartThings Developers. https://smartthings.developer.samsung.com/

[44] Aislan Tavares. 2021. QRGame. https://github.com/thisaislan/qrgame original-date: 2021-06-06T04:41:45Z.

[45] Thunkable. 2015. Thunkable: Build powerful, native mobile apps without coding. https://thunkable.com/

[46] VMware. 2006. What is VMware Horizon? | VDI Software. https://www.vmware.com/products/horizon.html

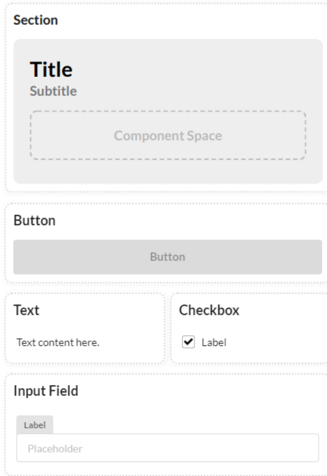
[47] Thomas Zachariah, Joshua Adkins, and Prabal Dutta. 2020. Browsing the Web of Connectable Things. In *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks* (Lyon, France) *(EWSN '20)*. Junction Publishing, USA, 49–60.

[48] Thomas Zachariah and Prabal Dutta. 2019. Browsing the Web of Things in Mobile Augmented Reality. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, USA) *(HotMobile '19)*. Association for Computing Machinery, New York, NY, USA, 129–134. https://doi.org/10.1145/3301293.3302359

[49] Deze Zeng, Song Guo, and Zixue Cheng. 2011. The Web of Things: A Survey (Invited Paper). *Journal of Communications* 6, 6 (Sept. 2011), 424–438. https://doi.org/10.4304/jcm.6.6.424-438

[50] Xiong Zhang and Philip J. Guo. 2019. Mallard: Turn the Web into a Contextualized Prototyping Environment for Machine Learning. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 605–618. https://doi.org/10.1145/3332165.3347936

A SUPPLEMENTARY FIGURES



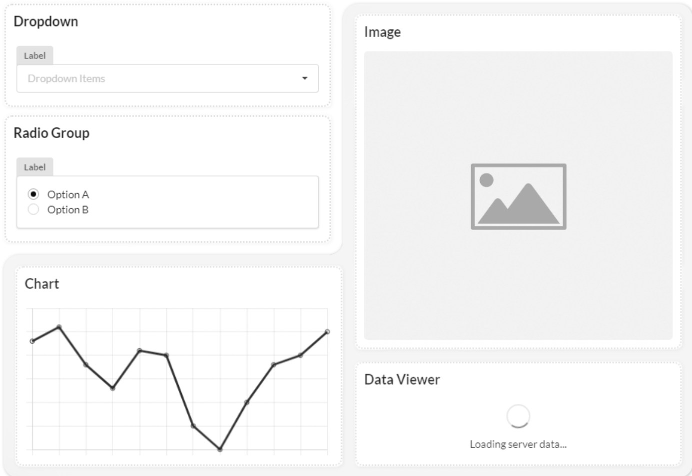


Fig. 10. UI components currently supported by enFrappé

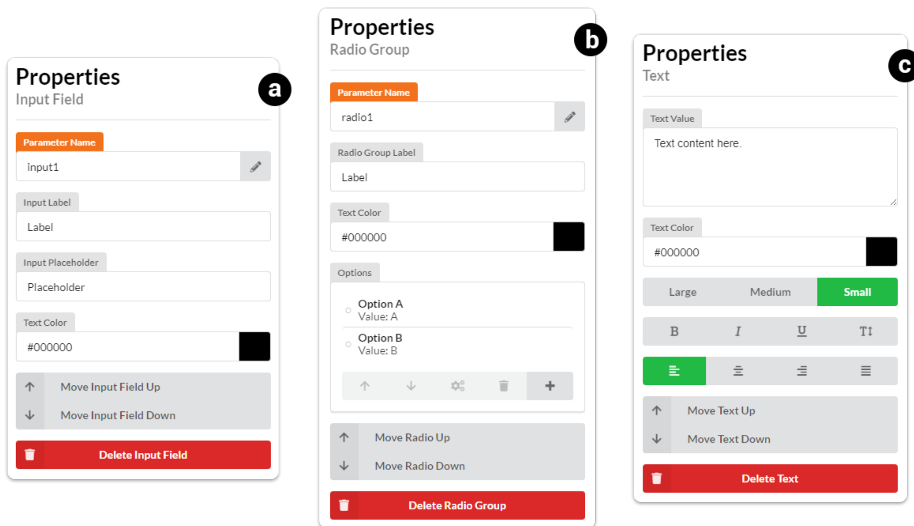


Fig. 11. Frappé decomposes every UI component into a set of basic parameters. (a) Properties of an input field component; (b) properties of a radio group component; (c) properties of a text component.

Received January 2023; Revised May 2023; Accepted June 2023