Meridian: A Design Framework for Malleable Overview-Detail Interfaces

Bryan Min
bdmin@ucsd.edu
University of California San Diego
La Jolla, California, USA

Haijun Xia
haijunxia@ucsd.edu
University of California San Diego
La Jolla, California, USA

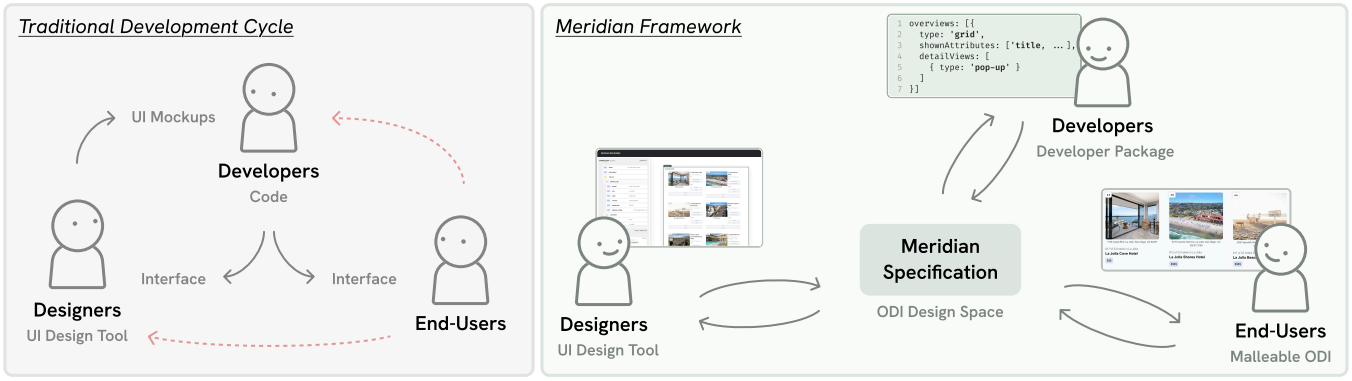


Figure 1: Meridian is a design framework for creating overview-detail interfaces (ODIs) that are malleable by default. Traditionally, designers and developers build interfaces using their respective tools, but this separation makes it difficult to transfer designs across teams. At the same time, this process often excludes end-users from personalizing the very interfaces intended for them. Meridian introduces a specification language for ODIs that integrates with developer packages, UI design tools, and end-user interfaces—enabling all three groups to equally establish interface needs.

ABSTRACT

Overview-detail interfaces (ODIs), which present an overview of multiple items alongside a detailed view of a selected item, are ubiquitously implemented in software interfaces. However, the current design and development pipeline lacks the infrastructure to easily support end-user customization, limiting its ability to support diverse information needs. This research envisions a development cycle for building malleable interfaces—one where designers, developers, and end-users alike can create, modify, and use the interface equally. To establish a foundation for this infrastructure, we introduce Meridian, a design framework for guiding and facilitating the creation of malleable ODIs. The framework consists of a high-level declarative specification language for ODIs as well as its tools, including a UI development package and a no-code web builder to facilitate the development and design of malleable ODIs. We demonstrate how Meridian supports designers, developers, and end-users alike can design, implement, and interact with ODIs in novel ways using their respective familiar tools and platforms. Finally, we discuss technical tradeoffs, potential solutions, and opportunities for enabling malleability for interfaces by default.

UIST '25, September 28-October 1, 2025, Busan, Republic of Korea

ACM ISBN 979-8-4007-2037-6/2025/09
https://doi.org/10.1145/3746059.3747654

CCS CONCEPTS

• **Human-centered computing** → **User interface toolkits**; *Interactive systems and tools.*

KEYWORDS

Overview-Detail Interfaces, Design Framework, Specification Language, Malleable Interfaces

ACM Reference Format:
Bryan Min and Haijun Xia. 2025. Meridian: A Design Framework for Malleable Overview-Detail Interfaces. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28-October 1, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3746059.3747654

1 INTRODUCTION

Overview-detail interfaces (ODIs) are among the most ubiquitous interface design patterns in information systems, as they support a fundamental information behavior—users often need to view an overview of large collections of information to identify ones of interest and then examine them in detail. ODIs can be found in our email clients, calendar, shopping websites, food delivery applications, and numerous others.

For decades, ODIs have been developed with a one-size-fits-all approach like many other software applications: developers configure a fixed overview (e.g., grid, map, timeline), determine which details to show in the overview, and define the composition and

interaction of the views. However, a single configuration fails to meet the diverse needs of all users [23, 40]. Consequently, users must work with overviews that do not match their mental models and repeatedly switch between views to gather details of interest, hindering the effective use of information. To address this friction, Min et al. proposed the notion of *malleable* ODIs, which enables end-users to flexibly manipulate the presentation of attributes in the views as well as their composition and layout [38]. For example, end-users can customize a malleable ODI in a shopping website by surfacing couch dimensions into the overview and then transforming the overview into various representations such as a color space to explore couches by color. Such malleability was found empowering and desirable, as it enabled end-users to flexibly manipulate the ODI to suit their own needs [38].

Despite the potential of malleable ODIs, the existing UI design and development approach lacks the infrastructure to easily support malleability in interfaces. First, designers must create mockups for many, if not all, potential transformations available on the interface, requiring them to create and manage numerous combinations of variations. Second, developers must implement these designs from scratch, requiring them to bind attributes to the UI, organize views and their content, build their navigation logic, and re-implement customization features that are already present in other interfaces. Lastly, even if end-users can customize them, there is no succinct way of observing these customizations by designers and developers.

While designers create ODI designs, developers implement ODIs in code, and end-users interact with them through the interface, they all perform their activities within their individual tools and languages. However, if we want malleability supported in interfaces by default, we must enable designers, developers, and end-users to equally establish an agreement on the information shown and interactions possible on the interface. To achieve this, we need a design convention that designers, developers, and end-users alike can easily adopt and share.

We propose ***Meridian***, a design framework for guiding and facilitating the creation of malleable ODIs across all three stakeholders. At its core, Meridian is powered by a specification language that formalizes the conceptual model of ODIs. This specification language integrates into a development package that renders Meridian specifications into malleable ODIs and a visual website builder to explore variations of ODI designs. By leveraging the Meridian specification, designers can easily export and share ODI designs to developers, developers can instantly render them in the interface, and end-users can succinctly communicate customizations as logs back to designers and developers. By bridging familiar workflows across designers, developers, and end-users under a shared specification language, we reduce the cost to build and transfer malleable ODIs between different stakeholders.

We took an evaluation-by-demonstration approach [31] to showcase how three groups of major stakeholders, designers, developers, and end-users design, implement, and interact with ODIs in novel ways using their respective familiar tools and platforms. The Meridian specification, development package, interface builder, and a gallery of examples are open-source[1] to invite broader collaboration and enable a thriving malleable ODI ecosystem.

[1]https://github.com/meridian-ui/meridian

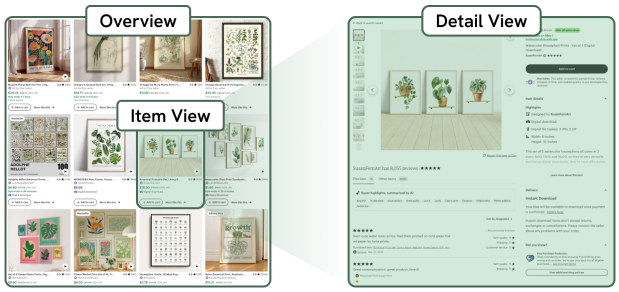


Figure 2: The overview-detail design pattern makes up three components: overview, item view, and detail view. This example shows Etsy's search results page presenting items in a grid and a detail view in a new page.

2 MALLEABLE ODI DESIGN CONVENTION

We aim to establish and promote a design convention of malleable ODIs through a design framework. We first define malleable ODIs, discuss the envisioned benefits of establishing malleable ODIs as a design convention, and then introduce our design approach.

2.1 Malleable ODI Definition

Malleable ODIs are extensions from overview-detail interfaces, which primarily involve *three* kinds of views (Fig. 2):

Overviews display a large collection of information entities using a specific organization structure to allow users to examine all the entities with the functional affordances of the structures. For example, an email client uses a list to organize emails by time, accommodation applications like Airbnb use a map for users to view homes by their location, and a scatterplot distributes data points by two number values.

Item Views represent information entities inside the overviews, and are positioned based on organizational rules of the overviews. Item views present key details that allow the users to gain an initial understanding of the information entities.

Detail Views contain all the details of an information entity. Detail views are typically invoked from the item views when users identifies and opens ones of interest.

Through an analysis of 303 ODIs found in existing websites, Min et al. identified three key design dimensions of ODIs [38]:

Content, describing which attributes are shown in each view.

Composition, describing the logical connections among views, such as how many overviews in the interface, and whether an overview opens one kind of pop-up or multiple.

Layout, describing the spatial arrangement of views within the interface and the interactions that invoke those views.

Following their analysis, they designed and developed *malleable* ODIs, ones that end-users can customize by transforming ODIs along variations in the three dimensions.

2.2 Design Approach

The goal of our design framework is to establish a convention for malleable ODIs that enables malleability by default, yet can easily integrate into familiar tools for designers and developers. Our approach is informed by the definition of malleable ODIs and the aforementioned benefits that we wish to obtain.

Employing High-level Declarative Specification. As indicated by the definition, ODIs can be modeled by their composition and content in the views, and both the composition and content can be described using properties of specific entities. This makes it suitable to describe ODIs with a high-level declarative specification. Therefore, the malleability of ODIs—in other words, modifications made on ODIs—can be expressed as transformations in the specification. As pointed out by prior work [21], a suitable high-level declarative language enables users to interact with key domain concepts more directly, and leave low-level execution to the run time.

Decoupling Data Binding and Views. Many existing development frameworks and UI libraries like Angular [16] and React [36] facilitate data-binding of attributes inside their views. This makes it challenging to flexibly interchange views and content for different UI variations or reuse the view of one ODI for another. We instead aim to define data binding and views separately when building ODIs. This would enable users to interchange data sources for the same ODI instance and interchange ODI instances for the same data source. It can also promote an ecosystem of ODI templates that can directly apply to diverse data sources.

Suitable Tooling for Different Stakeholders. Although a well-designed, high-level specification language that uses more readable formats (e.g., JSON) can lower the barriers of entry, they are still challenging to sift through. We aim to support our specification language with the necessary tools to support designers, developers, and end-users alike. This involves demonstrating end-user interaction techniques, a developer package, and a website builder.

3 THE MERIDIAN SPECIFICATION

We introduce the Meridian specification language for ODIs, which maps data attributes to the UI, controls which attributes are shown, composes overviews and detail views, defines their layouts, and enables control over which customizations the interface supports. Meridian specifications are structured as JSON objects to broadly integrate across various systems. A sample specification of a hotel booking website is shown and described in Figure 3.

3.1 Specifying Views

In its essence, an ODI's goal is to represent a collection of items such that it is approachable for the user. This process of representing them starts from specifying the views. Meridian defines two properties: `overviews`, which contains a list of overviews and item views, and `detailViews`, which contains a list of detail views. We describe how ODIs can be specified along the three dimensions using these views.

Overviews and Item Views. Meridian aims to simplify the creation of overviews by allowing users to define them using a single `type` property, which determines the overview's layout and representation of items, such as `'list'`, `'grid'`, and `'map'`. Each overview also specifies a default item view UI type through the `itemView` property, though users can explicitly define one as well. These specifications control the number of overviews displayed and specify which overview and item view UIs the ODI present. Meridian provides a set of default overview and item view types, which users can extend to include custom ones.

Detail Views. Detail views are also defined with a `type`, which includes sets of defaults such as `'basic'` and `'article'`. These views determine where and how the detail view appears through three properties: `openIn`, `openFrom`, and `openBy`. The `openIn` property specifies where the detail view opens, such as a `'pop-up'`, `'side-by-side'`, or `'new-page'`. The `openFrom` property determines which attributes, when selected, will open the detail view, and the `openBy` property defines the type of selection, such as `'click'` or `'hover'`. The `openFrom` property can be set to `'item'` to open the detail view upon selecting the entire item view, while `openBy` can support a custom event hook, allowing for custom ways to define how the user opens the detail view.

Detail views can be defined directly within an overview (Fig. 3.11) but can alternatively be defined in a dedicated list and linked to overviews by referencing the detail view `ids` (Fig. 3.12).

Shown and Hidden Attributes. Users need diverse sets of attributes depending on their tasks and views. To support this, Meridian allows users to specify lists of attributes to show or hide in each overview or detail view. By adding a role (roles explained in Section 3.2) to the `shownAttributes` or `hiddenAttributes` lists, users can determine which attributes are displayed or hidden in the views. Setting the property to `'all'` includes all attributes in that list, while setting it to an empty array includes none.

View-Specific Data Binding. Meridian also enables overviews and detail views to contain `bindingId` for each individual view. This allows, for instance, for multiple overviews to contain different sets of data, such as two different maps of hotels—one in San Francisco and one in La Jolla.

Recursion. There are often cases of overviews and detail views recursively composed of each other, a common example being a shopping page containing recommended items inside a detail view, which upon clicking a recommended item opens another detail view with another set of recommendations. Overviews and detail views can be composed recursively by referencing each other's `ids` in their respective view properties (Fig. 12).

3.2 Specifying Data Binding

Meridian provides the property `dataBinding` for binding data attributes to the ODI. This property is a single object that is responsible for mapping the data attributes of each item in a given data collection to the views in the ODI.

Attributes. To construct this data binding of data to ODI attributes, users can create an attribute object and reference a property in the data through the `value` property. The values are rendered as strings by default, but they can be defined as other types either provided by default such as `image` and `link`, or even a custom on created by the developer.

Roles. Since malleable ODIs render attributes into various view representations (e.g. lists, maps, timelines), one view could place an thumbnail image on the left, while another could place it on the right. To support placement-agnostic specifications from the data binding, we define attributes with *roles*. Roles embody a declarative approach to attribute placement, as it describes *what* an attribute should be rather than *where* it should be placed. This makes views

```
{
  dataBinding: [{
1   id: "query",
    binding: {
      itemId: ".itemId",
      attributes: [
2       value: ".name", roles: ["title"] },
        {
          value: ".details.photo",
3       type: "image",
          roles: ["thumbnail"]
        },
        { value: ".details.rating", roles: ["subtitle"], },
        {
          roles: ["subtitle"],
4       attributes: [
            { value: ".address_obj.street1" },
            {
              value: ".address_obj.street2",
5           condition: { exists: ".address_obj.street2" }
            },
            { value: ".address_obj.city" },
            { value: ".address_obj.country" },
          ]
        },
        { value: ".details.price_level", roles: ["tag"] },
        {
6       value: ".details.amenities",
          roles: ["spec"],
          transform: [{ map: ".amenity" }]
        },
      ],
    }
  }],
  overviews: [
    {
7   type: "list",
      itemView: { type: "compact" },
      shownAttributes: ["title", "subtitle", "thumbnail",
          "key-attribute", "action", "link"]
    },
    {
      type: "grid",
      detailViews: [
        {
8       type: "full",
          openFrom: ["title"],
          openIn: "side-by-side",
          openBy: "click",

9       overviews: [
            {
10            showIn: "footer",
              bindingId: "recommended",
              type: "carousel"
            }
          ],
          detailViews: [
            {
11          type: "gallery",
              openFrom: ["thumbnail"],
              shownAttributes: ["thumbnail"]
            }
          ]
        }
      ],
    },
  ],
  detailViews: [
12{ id: "article", type: "article" }
  ],
  malleability: {
13  content: {
      types: ["toggle"]
    },
    composition: {
14    disabled: true
    },
    layout: {
      types: ["menus"]
    }
  }
}
```

Data Binding Specification

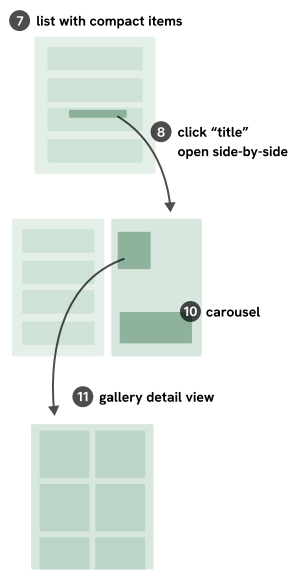
1. Reference the data source to data bind
2. Reference values from data with the prefix "."
3. Assign attributes with `type` to describe how they are rendered and `roles` map to where they are shown in the view.
4. Attributes can contain multiple attributes.
5. Conditionals can determine whether an attribute should or should not be rendered.
6. Attributes carrying lists of attributes can transform into at attribute group by mapping or filtering.

View Specification

7. Define the overview type. Other properties can additionally be defined such as the item view type and which attributes to show.
8. Detail views are defined inside overviews, and can be defined with a type, which elements to select to open the view, where to open it in, and how to select it.
9. Compose views in detail views.
10. Define where to show composed overviews. Overviews can also reference different data sources with `bindingId`, such as a list of recommended items.
11. Detail views can also specify which attributes to show or hide.
12. Define detail views independently from overviews. Overviews can reference external detail views via `id`.

Malleability Specification

13. Specify malleability features for each dimension, and how the features are shown in the UI.
14. All malleability features are enabled by default, but can be disabled through the specification.



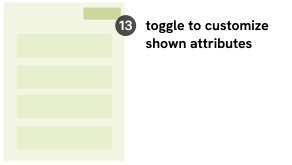


Figure 3: The Meridian specification is made up of three main components: data binding, views, and malleability. First, data binding describes the mapping from a data source to attributes rendered in the ODI. Second, views describe what attributes are shown in which views, how views are composed, layouts of items, and how detail views are opened. Lastly, malleability specifies which customizations the end-user will have access to.

responsible for placing roles, which are then populated by attributes assigned to those roles. For example, an attribute with the role `'title'` will be placed in the title area of a view, while an attribute with the role `'subtitle'` will be placed either above or below the title, depending on what the view has defined. Users can define roles to attributes and place the roles when creating their views, but Meridian also provides a standard with 12 roles to use by default.

Attribute Groups. There are many cases where closely related attributes to be grouped and styled together in the ODI. Meridian supports grouping by enabling attribute objects to contain either a data value or another list of attributes. Attribute groups share the same `style`, `role`, and `id` properties as standard attribute objects, allowing all attributes within a group to inherit a common UI mapping and style. For example, a hotel's address details may consist of multiple attributes. Users can define these attributes as a group, assign an `'address'` type and create a component that renders the attributes in a single row.

3.3 Specifying Malleability

The malleability specification controls two aspects: 1) whether to enable customization and, if so, for which dimensions, and 2) which UI interactions to provide for each customization option. For example, the UI for customizing which attributes to show or hide could be implemented as a `'toggle'`. When toggled on, this highlights all manipulable attributes, allowing users to select which attributes to show or hide in each view [38]. Another implementation could involve providing a `'console'` panel (Fig. 4) that lists all attributes, enabling users to check or uncheck which attributes they wish to show or hide.

4 DEMONSTRATION ONE: END-USERS

The Meridian specification is designed to be modifiable by end-users in powerful ways without having to expose the internal data source or codebase. For the first of our three sets of demonstrations, we illustrate Meridian's generative power by demonstrating how end-users can modify ODIs in ways that were previously non-trivial.

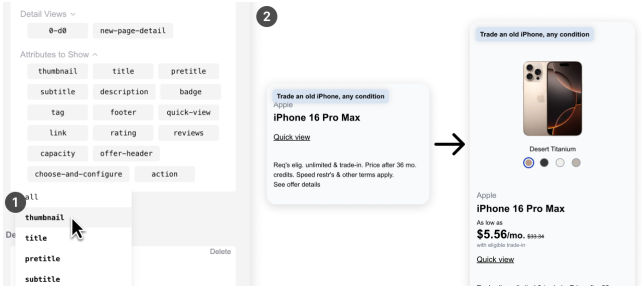


Figure 4: Meridian enables end-users to customize ODIs by directly modifying the specification. For instance, (1) selecting a "thumbnail" role to show in the overview will (2) directly transform item views to show "thumbnail" attributes.

Customization Widgets [D1-1]. Min et al. presented various interaction techniques for customizing ODIs along the three dimensions [38]. This included surfacing and hiding attributes in views, computing and generating new attributes on the fly, sorting and filtering by any attribute in the overview, creating multiple overviews, and transforming overview and detail layouts.

However, each design probe needed to implement each customization feature in each ODI dimension to provide all ODI customizations. In contrast, a Meridian-built malleable ODI can also support these interactions by default by presenting the specification in an editable console (Fig. 4). Indeed, websites can extend upon the console widget, such as by providing a global toolbar to prioritize overview layout customizations, tiled view managers for composition, or generative AI to prompt for customizations.

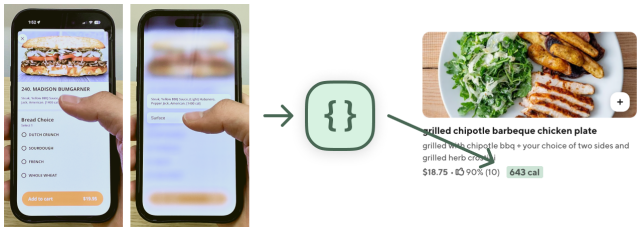


Figure 5: Systems can transport and reuse Meridian specifications to benefit end-users by automatically personalizing ODIs in different platforms and applications.

Exchanging Specifications [D1-2]. The Meridian specification is a JSON object, which makes exporting and importing it across malleable ODIs direct (Fig. 5). For example, end-users can directly copy specifications from websites and paste them into others. For instance, if a user prefers the overview layout of Etsy's over Amazon's, they can replace Amazon's overview with Etsy's while retaining Amazon's items. Additionally, a browser extension may also save frequently made ODI customizations and automatically apply them to other ODIs. For example, calorie-conscious users may consistently surface attributes relating to ingredients and calories, whereas those with cost concerns may surface prices instead. The browser extension can apply these customizations upon seeing new online menus, enabling each user to effectively carry a personal *lens* to view ODIs through their desired presentation.

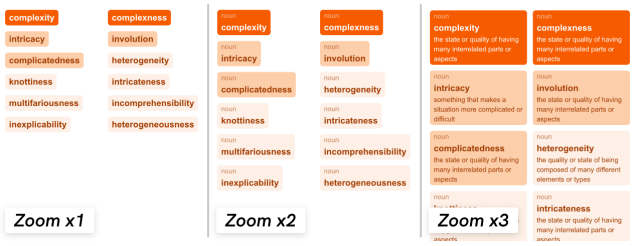


Figure 6: Meridian provides a simple solution to enabling semantic zoom on ODIs by translating scroll or pinch interactions to showing and hiding attributes in the overview.

Semantic Zooming [D1-3]. Semantic zooming is an interaction technique for zoomable user interfaces that scale items semantically as opposed to the traditional geometric approach [4]. This allows users to zoom out to view an overview of their canvas without losing view of textual descriptions of objects. This technique has been demonstrated across various spatial canvas systems throughout decades [10, 47, 48] demonstrating its broad use cases. However, it has found little implementation across real-world ODIs.

The Meridian specification enables interfaces to implement semantic zooming in any ODI in a simple implementation. Systems can detect mouse scroll or trackpad pinch events, and append or remove attributes from `shownAttributes` property of an overview. This allows users to zoom in and out to gradually reveal more or less attributes on any overview interface. For example, a user can semantic zoom in on a list of synonyms to reveal definitions and example sentences (Fig. 6).

Furthermore, Meridian can allow end-users to custom define levels of abstraction by allowing them to prioritize which attributes they want shown across different zoom levels. This contrasts to the traditional approach where the only the developers determine the most appropriate details to show at each level.

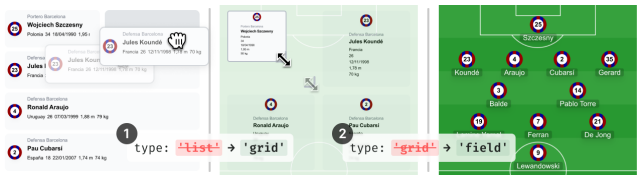


Figure 7: Meridian opens the space for exploring direct manipulation techniques for modifying ODIs. For instance, end-users can (1) drag an item to the side of its previous to transform the layout to a list and (2) resize an item into a small pin view to arrange items spatially.

Direct Manipulation [D1-4]. Min et al. demonstrated one opportunity for direct manipulation in malleable ODIs, where end-users could select and drag attributes between views. The Meridian specification provides a higher-level abstraction for operating on ODIs, enabling us to directly map direct manipulation interactions to moves from one ODI variation to another (Fig. 7).

First, dragging an item can translate to overview layout changes: dragging one item below another signals a shift to a list overview, whereas dragging an item to the right signals a change to a grid. Similarly, resizing an item can translate to layout changes: flattening an item vertically signals a shift into a table overview, while shrinking an item into a small square represents turning items into nodes within a spatial canvas layout. Second, while Min et al. demonstrated that dragging attributes signals the overview to display those attributes, we can extend this interaction such that dragging attributes while holding the "option" modifier key creates a new view containing only the dragged attributes. Lastly, when semantic zooming out on an overview (**D1-3**), users can select attributes to keep beforehand. As they zoom out, the interface gradually hides unselected attributes, but keeps selected ones visible.

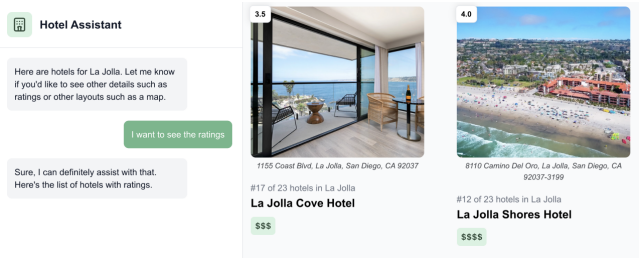


Figure 8: The Meridian specification language can serve as an interface for AI to generate ODIs. For instance, a hotel website could feature an AI chatbot that, when asked about hotels, generates a malleable ODI—customizable through either conversational input or direct interaction.

Generating with AI [D1-5]. Another opportunity Meridian presents is the ability to structure the generation of user interfaces with AI. AI can generate a Meridian specification directly rendered on the interface. One instantiation is demonstrated in Figure 8, where a hotel booking website can present an AI chatbot that facilitates hotel search by conversation alongside a Meridian-generated ODI on the side. Users can prompt AI to "show the ratings" or even ask to "show this list of hotels like how Airbnb shows their houses, prompting AI to reproduce Airbnb's overview layout, which presents a grid and a map.

Since ODIs generated with Meridian are scoped inside a single specification, AI applications can integrate Meridian into the interface in other diverse layouts. For example, interfaces can generate ODIs inside AI's chat response, similar how ChatGPT [41] and Gemini [15] display UI components inside the conversation.

5 DEMONSTRATION TWO: DEVELOPERS

In this section, we demonstrate how the Meridian framework can help developers build malleable ODIs. To do so, we built a developer package that renders Meridian specifications into interactive malleable ODIs on the web. We introduce the package by illustrating a development workflow and use the package to reproduce three malleable ODIs from real-world websites. Through these examples, we assess the breadth of the Meridian specification, sharing limitations and potential solutions to addressing them.

5.1 Developer Package

We use a hotel booking website as an example to demonstrate a workflow of the package. We use Tripadvisor's API[2] to ground our example with a real-world reference.

Create the attribute binding specification. Before using the package, the developer prepares API calls from their database that fetch a list of hotels and their details. Using this data, the developer adds each attribute they plan to display into the data binding specification. This includes the name of the hotel, images, location, ratings, reviews, rankings, amenities, and more. They then assign roles and types to attributes, such as an `'image'` type for the thumbnail.

[2]https://www.tripadvisor.com/developers

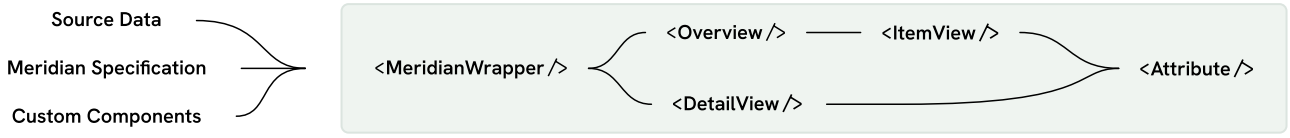


Figure 9: The developer package pipeline. Inputting the data, specification, and custom components into the developer package's main component, `<MeridianWrapper/>`, processes those inputs and renders them through the view and attribute components.

Input data and specification. Before creating the view specification, the developer passes the data and data binding specification into `<MeridianWrapper/>`, the base component that processes data, specifications, and custom components for the package to render (Fig. 9). By default, the Meridian package presents the items in a list layout, several attributes shown in the overview, and functionality for opening detail views by clicking either the hotel name.

Create the view specification. The developer makes additional adjustments by first creating an overview object and specifying which attributes to show in the overview with roles `'title'`, `'subtitle'`, `'thumbnail'`, `'tag'`, and `'review-summary'`. They also create a detail view inside the overview object that navigates to a new page instead of a pop-up. The package then instantly updates the views to support those changes. The developer decides to make another detail view that shows a gallery of all images of a selected hotel. To do so, they add a second detail view object with type `'gallery'` that opens by clicking the attribute with role `'thumbnail'`.

Customize ODI templates. The developer looks to apply their company's design system with branded color palettes, icons, and widget styles. The developer first copies the code of Meridian's view templates (i.e. `'list'` overview, `'profile'` item view, and `'full'` detail view) into their codebase and then customizes the layout and style of the attributes. They also add custom content such as text, labels, and icons. In addition to creating custom views, they also create custom attribute components such as a `'button'` and an `'image-gallery'`. Finally, they import their custom components into `<MeridianWrapper/>` to reflect their changes.

Test malleability. As the developer builds their malleable ODI by creating the specification and custom components, they interact with the ODI, opening detail views and customizing attributes and layout. If they find UI inconsistencies, the developer goes back to their specification and custom components to fix them.

5.2 Reproducing Real-World Examples

To assess the expressiveness of the Meridian developer package, we used it to reproduce three representative real-world ODIs: an online shopping page for smartphones[3], a soccer field with each team's formations[4], and a thesaurus that recursively composes overviews containing related words[5]. We outline our implementation strategies, limitations, and opportunities for growth.

Since the example websites were not open source, the first author recreated the data attributes from each page, saving them into JSON objects. They additionally recreated the theme of each ODI by saving image assets, replicating style with CSS, and developing necessary widgets (image gallery). Since these development responsibilities also exist in developing standard ODIs, we did not consider them towards challenges to build ODIs with our tools. We were able to reproduce the three ODIs to an extent at which they were fully interactive and presented all information from the original websites. We developed these websites using Next.js, a React framework. The gallery of the reproduced examples can be found at: https://github.com/meridian-ui/meridian.

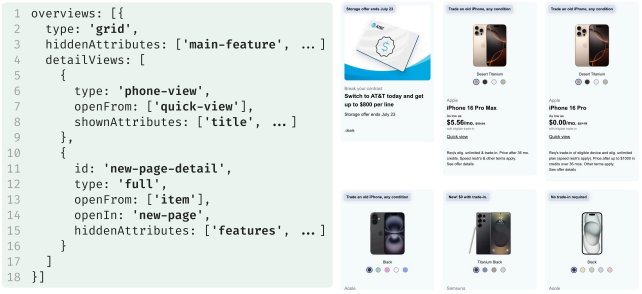


Figure 10: The Meridian view specification for AT&T's online phone shopping website.

5.2.1 ***Online Smartphone Shop [D2-1].*** The shopping page presented two different types of items: (1) a list of phones with many attributes including the brand, name, price, deals, and various spec options and (2) an advertisement for their company, which contained a different set of attributes. We specified these two types by creating two attribute groups, each with a condition that showed one group if the item's `type` was a phone or advertisement.

The ODI of the page presented a grid overview opening three different detail views: a new page showing the full view of a selected phone, a pop-up for a "quick view" of seeing key details, and another pop-up for a full list of rewards and offers for the phone. To specify this, we created an overview object with type `'grid'` and composed three detail view objects, each opening in their respective ways and with their respective set of attributes. Finally, we created custom grid, item, and attribute components so the ODI resembles the original website in terms of its attribute layout and UI style.

[3]https://www.att.com/buy/phones/
[4]https://www.marca.com/en/
[5]https://www.merriam-webster.com/dictionary/sophistication

```
overviews: [
  {
    id: 'players',
    type: 'soccer-field',
    bindingId: 'players',
    detailViews: [
      {
        type: 'player',
        openFrom: 'all'
      }
    ],
    hiddenAttributes: ['spec']
  }
]
```

Figure 11: The Meridian view specification for a soccer formation of players and their match statistics.

5.2.2 ***Soccer Formations [D2-2].*** This example presented a unique overview that presented two teams in various formations on a soccer field. Clicking a player opens a pop-up, presenting match information such as goals scored and minutes played. The overview uniquely places players depending on the team formation (e.g., 4-4-2, 4-2-3-1). To enable this, we added each player's `positionId` in the list of `internalAttributes`, since this data was not shown in any view and only used for positioning. We created a custom overview that places players depending on their team's formation and their `positionId`, and added the rest of the data binding specification, view specification, and other custom components.

```
detailViews: [
  {
    id: 'word',
    type: 'thesaurus',
    openFrom: 'all',
    shownAttributes: 'all',
    overviews: [ {
      id: 'synonyms-list',
      type: 'grid',
      itemView: { type: 'synonym' },
      showIn: ['spec'],
      shownAttributes: ['key-attribute'],
      attributeBindingId: 'synonyms',
      detailViews: ['word']
    } ]
  }
]
```

Figure 12: The Meridian view specification for Merriam-Webster's online thesaurus.

5.2.3 ***Thesaurus [D2-3].*** Our third example demonstrated two unique properties: (1) the detail view was the first interface as opposed to the overview, and (2) the overviews and detail views composed each other recursively. To support the first property, we specified the `overviews` property with an empty array and populated the `detailViews` property with one detail view object. To support the second, we first specified the detail view object with the id `'word'` and added an overview object into the detail view's `overviews` list. We then added the id `'word'` in the overview object's `detailViews` property, allowing the overview to open the detail view from which it originated. Finally, we assigned the overview two additional properties to enable the recursive ODI. Then, we specified the property `attributeBindingId` with value `'synonyms'`, which meant the overview presented a list of items inside an attribute, rather than a data binding specification. For this case, the overview presents the list of synonyms for a particular item. We first specified a `showIn` property to define where the overview appears within the detail view.

5.3 Malleable ODI Rendering Pipeline

5.3.1 Summary. The package renders ODIs in three main stages: (1) processing data, the specification, and custom components, (2) constructing the specified ODI variation, and (3) rendering the UI with given data attributes. First, to process data and the specification, a dedicated processing module maps the list of items from the data source into the structure of the data binding specification. This creates a list of items, each containing attributes specified by the data binding. Custom components are saved to the package's store, ready to render then when it identifies the type from the specification. Second, the package identifies the necessary types for overview, detail view, item view, and attribute components, such as the grid in Figure 10 or the popup in Figure 11. These view components are selected, ready to render to the interface. Finally, the data attributes from the data binding specification are passed down through the constructed view components and rendered in the attribute component where its specified type (e.g., string, image, link) is determined (Fig. 9).

Of the three detail views, the first one opens by clicking the entire item, so the item view enables a mouse click event that renders that detail view. The other two open by clicking an attribute, so the attribute associated to those views enable the click event.

5.3.2 Unique Cases. Our three examples, surface several unique cases that do not necessarily follow a basic ODI specification structure in which a single list overview opens a single detail view with several attributes shown.

The first example demonstrates how multiple details can open from one overview. This example contains three detail views. The first opens when the entire item component is clicked, so the item view attaches a click event listener that renders the detail view on click. The other two open when specific attributes in the overview are clicked, and each attribute gets its own click event listener.

The second example presents a custom soccer field overview, but the item views in this overview are icon-sized components, as opposed to larger item views in list and grid views. This means this overview has a maximum number of attributes it can show without cluttering view with too many. To address this, we updated our custom item view component to limit the number of attributes it could present with four attributes—one for each corner. By doing so, surfacing attributes did not just add the attribute to the overview, but also replaced another existing attribute, maintaining a compact size yet enabling customizability.

The third example demonstrates a recursive ODI, where an overview is inside its own detail view. The package supports recursive views by treating composed overviews as custom attribute components—similar to how buttons and image galleries are treated as custom attributes. To achieve this, the package first identifies the view that references its ancestor view—in this case, the overview. Then, before processing any parts of the specification, the processing module creates a new attribute in the data binding specification with type `'overview'`, adds the values from `showIn` into the attribute's `roles`, and transfers the attribute data referenced by `attributeBindingId` to the new attribute. This makes the overview attribute component render inside the layout of the detail view as opposed to being rendered as the view itself.

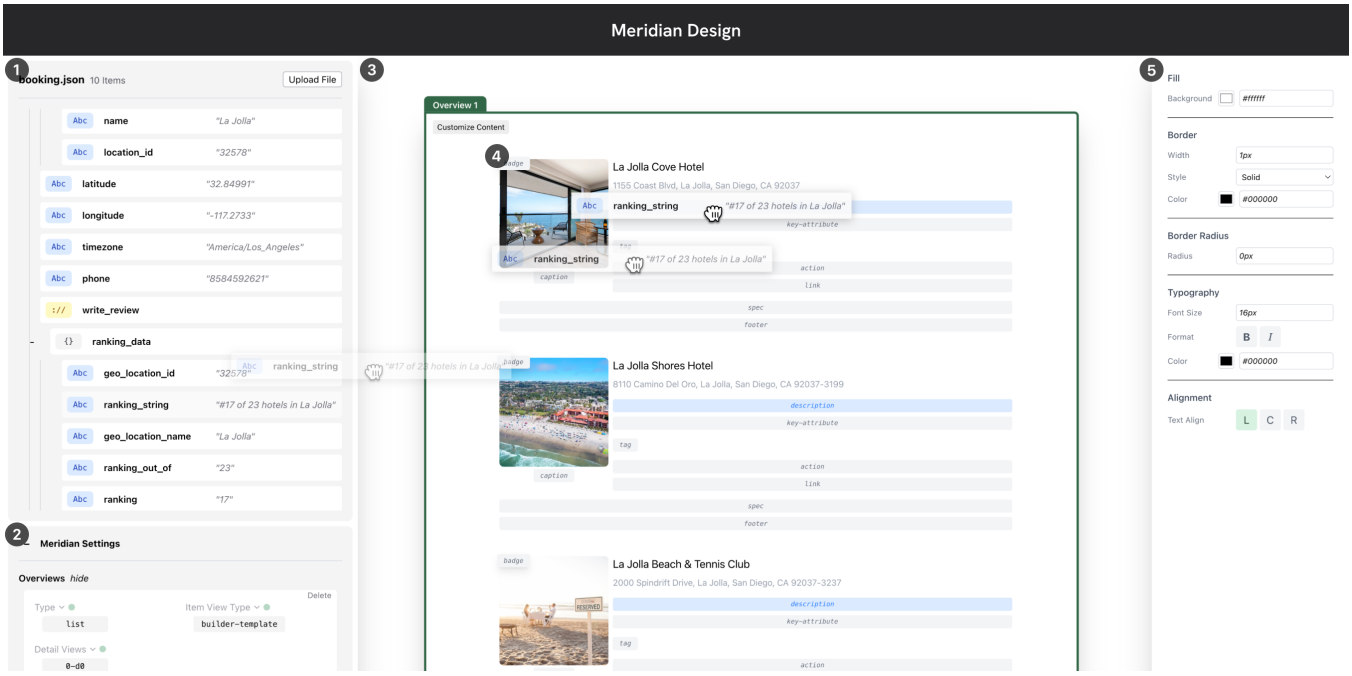


Figure 13: The Meridian no-code website builder interface. (1) Users can first import JSON or CSV data to drag attributes to the ODI, (2) modify the Meridian specification, (3) explore design variations on a spatial canvas, (4) either add attributes to template placements or manually position attributes, and (5) style them.

5.4 Limitations

As we developed the three examples, we faced several limitations in implementing aspects of the interface across components and the specification.

Imprecise attribute arrangements. Although the development package automatically arranged attributes across different view layouts, we inevitably needed to make precise changes in most views. For instance, although our default vertical item view could arrange attributes below the thumbnail vertically just as the smartphones example did, it did not provide a default option for attributes that floated on the right side. While our package offers an escape hatch through custom components, we see opportunities to improve expressiveness by supporting richer layout specifications.

Limited support for niche conditional cases. We encountered several conditional cases in our specification we could not elegantly address. In the smartphones example, a detail view presented an attribute group containing a list of offers for purchasing a phone, while the overview presented only the first one. Although we can specify which attributes to show, we do not yet support attributes to show depending on their index or order in the list. Our workaround was to create another attribute that only contained the first element in the list of offers. In future iterations of the specification, future iterations should support more niche conditions as these ones.

Limited support for variable number of views. While Meridian supports an arbitrary number overviews and detail views in the specification, it does not support a *variable* number of views. In the thesaurus example, some words carry multiple definitions, each presenting an overview list of synonyms. Although the data binding specification could support this case, the view specification could not. Future iterations of Meridian should explore how views can be mapped to a variable number of overviews, possibly by supporting map and transform operations in the overview.

6 DEMONSTRATION THREE: DESIGNERS

In this section, we demonstrate how Meridian can facilitate unique design workflows for creating malleable ODIs. We used the developer package described in the previous section to build a no-code website builder (Fig. 13) that allows designers to import data, bind attributes to the ODI, style them, explore a design space of ODI variations, and integrate data analytics to inform their design choices.

6.1 No-Code Website Builder

Many interface and visualization authoring tools—such as Data Illustrator [34], Voyager [56], Lyra [58] for visualizations, and Wix [55] and Squarespace [45] for websites—have demonstrated the usability and expressive power of their platform for authoring complete visualizations and websites without code. To demonstrate how Meridian can do the same for ODIs, we designed our no-code website builder to resemble these tools, adopting similar interaction paradigms to support ease of use and expressiveness. We illustrate the website builder through a scenario of designing a hotel booking website, highlighting three unique design workflows that leverage the Meridian specification.

6.1.1 ***Building from Scratch [D3-1].*** Designers can borrow an instance of fetched hotel data from a developer and import the file into the web builder. Upon importing the file, a list of attributes populate on the left sidebar. They can then explore the attributes, assigning types if needed.

The center of the interface presents a canvas with template overviews and detail views, which display a default arrangement of attribute roles. Designers can drag attributes from the left sidebar onto one of these roles, populating that role with the attribute. Then, they can select attributes and change their style using the style configuration features provided on the right sidebar. Designers can also rearrange attributes in the item view. They start by selecting an individual attribute, unlocking its position using the right sidebar, and then dragging it to a new location. Additionally, they can move an entire group of attributes associated with a specific role to the bottom of the item view.

Designers can then modify to the ODI design. They start by adjusting which attributes are visible in the overview, then create a new detail view that reveals the full address of a hotel when the user hovers over the location. Finally, they can export their ODI design specification as a JSON object and share it to developers for them to integrate into the codebase.

6.1.2 ***Generating and Viewing the Design Space [D3-2].*** Meridian's specification, which formalizes the ODI design space, unlocks an opportunity for designers to explore a large design space of ODI variations, facilitating divergent thinking [19] and parallel design [30]. The web builder provides this opportunity by presenting a design space of ODI variations on a canvas (Fig. 14).

To view the design space, the designer navigates to the left sidebar with ODI settings and clicks "Design Space" next to a property. Upon clicking this button for the `type` property in the overview, the overview in the canvas multiplies and spreads into a grid, showing all of the overview layout types. The designer looks through ones of interest, while dragging a few more attributes into the overview, displaying the attributes for every item across every overview. Finally, they select a variation to set as their default.

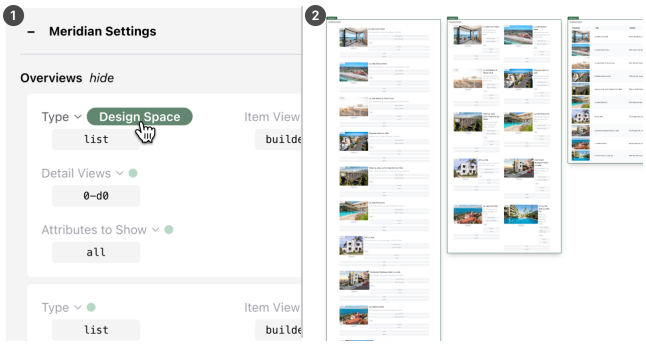


Figure 14: Meridian allows designers to explore the larger design space of ODIs by generating ODI variations on the canvas. Users can (1) click "Design Space" on a property and (2) view and interact with the produced variations.

6.1.3 ***Data-driven Design [D3-3].*** While iterating on our design framework, we conducted informal interviews with industry designers and developers across diverse sectors (contracting, banking, SaaS, etc.), gathering insights on their ODI creation workflows. Some interviewees shared that their design teams struggled to justify certain design choices—such as using a dropdown versus navigating to a new page for a detail view—hinting at a lack of detailed analytics on user behavior with ODIs.

Although we currently lack a comprehensive and precise understanding of which ODI variations benefit which scenarios, Meridian presents an opportunity to facilitate this process of gathering user interactions structured by the specification, translating them into concrete design insights, and presenting them to the designer.

In the web builder, the designer may receive an additional dataset from data analysts containing insights on time spent interacting with variations of a malleable ODI, such as a popularity distribution of various attributes, time spent on certain layouts, and popular ODI settings created and used across large populations. Upon importing this dataset, the web builder may highlight popular variations with colors and shades, as well as recommend combinations (Fig. 15). This would allow the designers to create better default ODI instances for users, such as moving certain attributes to the overview, or moving attributes in easily discoverable areas in the detail views.

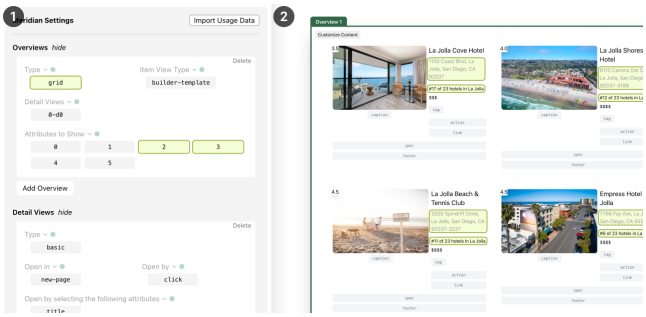


Figure 15: The Meridian specification can map to usage analytics, providing insights into how popular various ODI customizations are. Importing a usage data file into the website builder can highlight popular ODI variations in both (1) the settings console and (2) the interface mockups.

6.2 Implementation

We implemented the no-code website builder as a Next.js web application. Under the hood, the website builder is built around the Meridian developer package, adding various UI design features wrapped around the ODI. As depicted in Figure 13, these features include a data-binding component, settings panel, a spatial canvas, and a UI style editor. The ODI components are rendered inside the spatial canvas.

The main extension of the website builder is our custom-built, *editable* item component. Instead of displaying default item view components that display the attributes, editable item views enable users to drag and drop attributes onto the component (Fig. 13.4) and directly select, edit, and style them. We surprisingly found that this approach opened two interaction opportunities for the

no-code web builder. First, we could create multiple *modes* for editing item attributes. For instance, we implemented one mode for dragging attributes into a template that automatically places them into fixed layouts (shown in Figure 13.4) and another mode for flexibly placing attributes anywhere without layout constraints. This helped manage a balance between too much structure and too little when creating UI mockups. Second, switching between item views offered the opportunity to switch between *editing* and *previewing* the malleable ODI. Since the editing features are scoped inside the item view component, the settings panel (Fig. 13.2) offered both the editable and default item views for users to switch between.

Another key extension of the website builder is the end-user interface for data binding attributes to the interface. In the current Meridian specification language, data binding attributes and adding roles to each attribute may grow cumbersome and difficult to manage in the developer package, especially for detail-rich ODIs. Dragging and dropping attributes from the attributes panel (Fig. 13.1) to the editable item view (Fig. 13.4) automatically completes two actions that involve data binding. First, the system creates a new attribute object from the dropped attribute and is appended to the `attributes` list in the specification (Section 3.2). Next, when the attribute is dropped in one of the areas in the template (Fig. 13.4), the system adds the name of the area to the attribute object as a role (Section 3.2). From our internal tests with the website builder, we found that this could reduce the cost of data binding attributes to the interface, potentially reducing this burden on the developers.

7 DISCUSSION AND FUTURE WORK

We presented Meridian, a design framework that aims to set a convention for designing, building, and interacting with malleable ODIs alongside tools to enable this convention. Meridian is powered by the Meridian specification language, acting as a shared language between designers, developers, and end-users. We demonstrated the expressive power of Meridian by implementing malleable ODIs and tools that facilitate novel workflows for end-users, developers, and designers.

Real-world use of Meridian. We implemented malleable ODIs using real data and real websites, showcasing Meridian's ability to support various ODIs for real-world interfaces. However, building and designing software interfaces involves understanding and integrating a much larger scope than we explored in this paper.

Many production-grade software systems typically provide two REST API calls, one for the overview and one for the detail view. As a result, making a customization such as surfacing a detail view attribute to the overview would require the system to fetch each detail view's data, incurring significant performance costs. However, this architecture is not just a challenge for Meridian, but also for existing ODI systems. One of our interviewees (Section 6.1.3), a front-end engineer building ODIs with REST APIs, mentioned in order to change which attributes are shown in a view, they need to wait for their back-end engineers to update the API to implement those changes.

One avenue that systems may shift towards when using the Meridian design framework is a more flexible API querying method like GraphQL [6] over the more traditional, table-centric REST APIs. Another interviewee, a full-stack engineer who developed apps using GraphQL at a SaaS company, noted that GraphQL effectively addresses the problem in REST APIs by batching API calls for required attributes. However, it can introduce challenges in efficiently caching these attributes and managing fetches with varying costs.

We see an opportunity to use Meridian to further explore the tradeoffs involved in implementing malleable ODIs at an architectural level. Even though some teams inevitably may choose not to build ODIs with Meridian, there are still groups that can benefit from the framework without needing to engage with the technical details of the software, such as e-commerce websites that rely on CSV data sources to manage their inventory. In future work, we aim to deploy, test, and observe the real-world use of Meridian across various settings and system architectures, as well as gather insights from long-term end-user behavior.

Customizations as first-class interactions. Our demonstrations showcased various interaction techniques that could modify the Meridian ODI specification, including semantic zooming, direct manipulation, and prompting with AI. Although the interaction techniques are not generally considered customization techniques, mapping them to changes in the Meridian specification enables end-user to customize ODIs. We see a parallel with sorting and filtering, which are customization operations, now ubiquitous across many interfaces and considered standard ways to interact with a collection of items. As Meridian makes ODI customizations more broadly available, we are interested in exploring how user interactions shift towards the customization features, and their mindset towards performing these customizations. We seek to further study this co-adaptive phenomenon [35] with malleable ODIs with Meridian.

Integrating Meridian into more tools. Our demonstrations explored various tools for developers and designers to build malleable ODIs. We implemented a developer package to help specify the data binding and views of ODIs and built on top of this a website builder for allowing designers to design ODIs without code. We see opportunities for Meridian to be used in more workflows, through more tools. For instance, continuous integration and development (CI/CD) teams may want to present visualizations according to usage behavior of ODIs. We envision visualization tools can leverage the Meridian specification and usage behavior to form interactive ODI variations mapped to visualizations Additionally, UI component managers like Storybook help designers and developers manage responsive components and widgets they can use to help integrate a design system into their applications. Such tools may further integrate the Meridian specification to allow users to easily manage custom overview, item view, detail view, and attribute components. They can further create responsive designs of these components by binding attributes such as width and height to different specifications. This may enable desktop users to open detail views in a side-by-side view, while mobile users open detail views in a new page.

[6] https://graphql.org/

8 RELATED WORK

8.1 User Interface Frameworks

A design and development framework for user interfaces establishes a *convention* for creating a specific type of interfaces and often comes with *tools*, such as software libraries, as well as design and development environments to help the designers and developers follow the convention.

User interface conventions are often defined through specification languages, such as HTML and CSS for web structure and styling, or SwiftUI for iOS applications [2]. Prior work has also explored JSON-based specifications for web interfaces, mapping data attributes to rendered UIs [7, 9, 44]. These conventions have been carried over into tools that help design, manage, and build these interfaces. For example, Figma [11] allows users to export the styles of UI mockups created on its platform as CSS code, while Storybook [46] helps manage design systems for UI components built with HTML and CSS. UI frameworks like Material-UI [17], Ant Design [18], and Bootstrap[6] establish conventions for styling buttons, sliders, forms, and pop-ups by implementing them as reusable UI components for web applications. Additionally, platforms like Wix [55] and Webflow[52] provide a visual editor for building HTML/CSS websites through drag-and-drop interactions. While these design frameworks support the creation of both full websites and individual UI components, Meridian sits in between these layers. Meridian's tools can integrate into broad design and development platforms like Figma and Wix, while also leveraging UI components from libraries such as Material-UI and Bootstrap.

An analogy can be drawn between the theories and frameworks of user interfaces with those for visualizations. In visualization, programming libraries like D3.js [8] allows developers to create any data-driven graphics, by enabling the developers to manipulate the DOM elements and establish data bindings. At a higher-level of abstraction, Wilkinson's Grammar of Graphics introduced the conceptual framework for thinking about how to describe and build plot-based visualizations [54]. With this conceptual framework, plotting systems like ggplot2 [53] and Vega-Lite [43] enable users to create data plots using high-level declarative language, without concerning about how low-level graphical elements are composed. While systems like Vega-Lite does not have expressive power to create any visualizations, it still enables an ecosystem of tools as it eased the development of commonly used data charts. Vega-Lite offers broad extensibility and integration into various tools, enabling users to author visualizations without code [58], explore and discover relevant visualizations [56], make systematic design choices [39], extend to other platforms like Python [51], and expand the specification to support responsiveness or animation [26, 59].

We see Malleable ODIs residing at the level of ggplot2 [53] and Vega-Lite, but for user interfaces, targeting one of the most ubiquitous interface design patterns. The presented work is a first step towards establishing an ecosystem of diverse tools to support malleable ODIs for diverse platforms and usage scenarios.

8.2 Achieving the Vision of Malleable Interfaces

Generations of HCI researchers have envisioned digital information environments that are personal, dynamic, and malleable—ones users can expressively customize to achieve their own needs through new functionalities and representations [12, 23, 24, 28]. This perspective compares software to "clay"—a material in which end-users can dynamically shape to address their desired needs [24]. A common approach in malleable interfaces has been to open underlying data models and code, allowing users to create custom scripts in the GUI [3], modify the look and behavior of objects [25], customize the presentation of views [1], and create custom representations of web pages and media [14, 28]. Another approach explored opened existing systems, particularly browsers, to enable end-users to directly customize CSS properties [27, 29, 50], embed scripts and annotations into the DOM [42], transform list items with custom operations [22, 33], and form web mashups [5, 13, 20, 32, 37, 49, 57].

These approaches have explored making the underlying architecture more open to malleability. The first approach enables malleability inherently through the software architecture. However, opening the software can be challenging for end-users who are not familiar with programming. The second approach enables malleability by tackling one feature or aspect at a time, such as the style of a website through an interface for customizing CSS or the order of items with a function to scrape web elements and sort them. However, the focus on only one feature of the interface limits end-users from seeing a design space of potential customizations they can make, and limits them from transferring what they have learned in one context to another.

Instead, we explore how to make interfaces malleable one design pattern at a time, starting with the overview-detail pattern. By identifying key design dimensions and making them directly manipulable, we enable users to expressively customize the pattern without requiring programming skills. Moreover, focusing on a design pattern allows users to easily transfer what they learn from one instantiation to another.

Perhaps the most major barrier to adopting malleable interfaces is motivating the development of malleable systems in the first place. Compared to approaches that require adopting entirely new programming models [7], ODIs are relatively self-contained components that can be more readily integrated into existing UI development workflows. While ODIs alone do not make entire applications malleable, their strength lies in their ubiquity. By establishing design conventions for malleable ODIs and providing supporting tools for their design and development, we aim to drive widespread improvements in the malleability of software interfaces.

9 CONCLUSION

This paper introduces Meridian, a design framework that guides and facilitates the creation of malleable ODIs. Meridian is powered by the Meridian specification language which describes ODIs along three dimensions: content, composition, and layout. Meridian provides a shared language across different user groups of ODIs, enabling end-users to customize ODIs, developers to build ODI websites, and designers to create ODI designs, all using the Meridian specification language. We demonstrated how Meridian can integrate into familiar platforms and tools used by the three user groups and how they can interact with malleable ODIs in novel ways. We hope our work illustrates the opportunities for an ecosystem of malleable ODIs through Meridian.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank Peiling Jiang, Brian Hempel, and Matthew T. Beaudouin-Lafon for their insightful feedback on the design of the specification language. This work was supported by NSF under grant IIS-2432644.

REFERENCES

[1] Eytan Adar, David Karger, and Lynn Andrea Stein. 1999. Haystack: per-user information environments. In *Proceedings of the Eighth International Conference on Information and Knowledge Management* (Kansas City, Missouri, USA) *(CIKM '99)*. Association for Computing Machinery, New York, NY, USA, 413–422. https://doi.org/10.1145/319950.323231

[2] Apple. 2025. SwiftUI. https://developer.apple.com/xcode/swiftui/ Accessed April 2, 2025.

[3] Bill Atkinson. 2024. HyperCard. https://arstechnica.com/gadgets/2019/05/25-years-of-hypercard-the-missing-link-to-the-web/. Accessed December 3, 2024.

[4] Benjamin B. Bederson and James D. Hollan. 1994. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology* (Marina del Rey, California, USA) *(UIST '94)*. Association for Computing Machinery, New York, NY, USA, 17–26. https://doi.org/10.1145/192426.192435

[5] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and customization of rendered web pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology* (Seattle, WA, USA) *(UIST '05)*. Association for Computing Machinery, New York, NY, USA, 163–172. https://doi.org/10.1145/1095034.1095062

[6] Bootstrap. n.d.. Bootstrap. https://getbootstrap.com/ Accessed: 2025-04-10.

[7] Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandsted Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 492, 20 pages. https://doi.org/10.1145/3491102.3502064

[8] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). http://vis.stanford.edu/papers/d3

[9] Yining Cao, Peiling Jiang, and Haijun Xia. 2025. Generative and Malleable User Interfaces with Evolving Task-Driven Data Schema. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '25)*. Association for Computing Machinery, New York, NY, USA, 21 pages. https://doi.org/10.1145/3706598.3713285

[10] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. 2009. A review of overview+detail, zooming, and focus+context interfaces. *ACM Comput. Surv.* 41, 1, Article 2 (jan 2009), 31 pages. https://doi.org/10.1145/1456650.1456652

[11] Figma Inc. 2016. Figma. https://www.figma.com/. Accessed: 2025-04-08.

[12] Amy Rae Fox, Philip Guo, Clemens Nylandsted Klokmose, Peter Dalsgaard, Arvind Satyanarayan, Haijun Xia, and James D. Hollan. 2020. Towards a dynamic multiscale personal information space: beyond application and document centered views of information. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(Programming '20)*. Association for Computing Machinery, New York, NY, USA, 136–143. https://doi.org/10.1145/3397537.3397542

[13] Giuseppe Ghiani, Fabio Paternò, Lucio Davide Spano, and Giuliano Pintori. 2016. An environment for End-User Development of Web mashups. *International Journal of Human-Computer Studies* 87 (2016), 38–64. https://doi.org/10.1016/j.ijhcs.2015.10.008

[14] Camille Gobert and Michel Beaudouin-Lafon. 2023. Lorgnette: Creating Malleable Code Projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 71, 16 pages. https://doi.org/10.1145/3586183.3606817

[15] Google. 2025. Google Gemini. https://gemini.google.com/ Accessed July 12, 2025.

[16] Google. n.d.. Angular. https://angular.dev/ Accessed: 2025-04-10.

[17] Google. n.d.. MaterialUI. https://mui.com/ Accessed: 2025-04-10.

[18] Ant Group. n.d.. Ant Design. https://ant.design/ Accessed: 2025-04-10.

[19] Joy Paul Guilford. 1961. Three faces of intellect. (1961). https://doi.org/10.1037/h0046827

[20] 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* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1294211.1294254

[21] Jeffrey Heer and Michael Bostock. 2010. Declarative language design for interactive visualization. *IEEE transactions on visualization and computer graphics* 16, 6 (2010), 1149–1156.

[22] David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling web browsers to augment web sites' filtering and sorting functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (Montreux, Switzerland) *(UIST '06)*. Association for Computing Machinery, New York, NY, USA, 125–134. https://doi.org/10.1145/1166253.1166274

[23] David Karger. 2007. Haystack: Per-User Information Environments Based on Semistructured Data. In *Beyond the Desktop Metaphor: Designing Integrated Digital Work Environments*, Victor Kaptelinin and Mary Czerwinski (Eds.). MIT Press, start–end.

[24] Alan Kay and Adele Goldberg. 1977. Personal dynamic media. *Computer* 10, 3 (1977), 31–41.

[25] Alan C. Kay. 1993. The early history of Smalltalk. *SIGPLAN Not.* 28, 3 (March 1993), 69–95. https://doi.org/10.1145/155360.155364

[26] Hyeok Kim, Ryan Rossi, Fan Du, Eunyee Koh, Shunan Guo, Jessica Hullman, and Jane Hoffswell. 2022. Cicero: A Declarative Grammar for Responsive Visualization. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 600, 15 pages. https://doi.org/10.1145/3491102.3517455

[27] Tae Soo Kim, DaEun Choi, Yoonseo Choi, and Juho Kim. 2022. Stylette: Styling the Web with Natural Language. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 5, 17 pages. https://doi.org/10.1145/3491102.3501931

[28] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 280–290. https://doi.org/10.1145/2807442.2807446

[29] Google Chrome Labs. 2024. Project VisBug. https://github.com/GoogleChromeLabs/ProjectVisBug/. Accessed August 29, 2024.

[30] Paul Laseau. 2000. *Graphic thinking for architects and designers*. John Wiley & Sons.

[31] 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

[32] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user programming of mashups with vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces* (Sanibel Island, Florida, USA) *(IUI '09)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/1502650.1502667

[33] Geoffrey Litt, Daniel Jackson, Tyler Millis, and Jessica Quaye. 2020. End-user software customization by direct manipulation of tabular data. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 18–33. https://doi.org/10.1145/3426428.3426914

[34] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John Stasko. 2018. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. 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, 1–13. https://doi.org/10.1145/3173574.3173697

[35] Wendy E. Mackay. 1990. *Users and Customizable Software: A Co-Adaptive Phenomenon*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA. https://dspace.mit.edu/handle/1721.1/14087

[36] Meta. n.d.. React Documentation. https://react.dev/ Accessed: 2025-04-10.

[37] Bryan Min, Matthew T Beaudouin-Lafon, Sangho Suh, and Haijun Xia. 2023. Demonstration of Masonview: Content-Driven Viewport Management. In *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23 Adjunct)*. Association for Computing Machinery, New York, NY, USA, Article 60, 3 pages. https://doi.org/10.1145/3586182.3615827

[38] Bryan Min, Allen Chen, Yining Cao, and Haijun Xia. 2025. Malleable Overview-Detail Interfaces. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '25)*. Association for Computing Machinery, New York, NY, USA, 26 pages. https://doi.org/10.1145/3706598.3714164

[39] Dominik Moritz, Chenglong Wang, Gregory Nelson, Halden Lin, Adam Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2019). https://doi.org/10.1109/TVCG.2018.2865240

[40] Bonnie A. Nardi. 1993. *A small matter of programming: perspectives on end user computing*. MIT Press, Cambridge, MA, USA.

[41] OpenAI. 2025. Introducing ChatGPT Search. https://openai.com/index/introducing-chatgpt-search/ Accessed July 12, 2025.

[42] Hugo Romat, Emmanuel Pietriga, Nathalie Henry-Riche, Ken Hinckley, and Caroline Appert. 2019. SpaceInk: Making Space for In-Context Annotations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 871–882. https://doi.org/10.1145/3332165.3347934

[43] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[44] Yoshiki Schmitz. 2025. https://x.com/yoshikischmitz/status/1176642448077967362 Accessed April 2, 2025.

[45] Inc. Squarespace. n.d.. Sqarespace. https://www.squarespace.com/ Accessed: 2025-07-13.

[46] Storybook. n.d.. Storybook: Fontend Workshop for UI Development. https://storybook.js.org/. Accessed: 2025-04-09.

[47] Sangho Suh, Meng Chen, Bryan Min, Toby Jia-Jun Li, and Haijun Xia. 2024. Luminate: Structured Generation and Exploration of Design Space with Large Language Models for Human-AI Co-Creation. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 644, 26 pages. https://doi.org/10.1145/3613904.3642400

[48] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecape: Enabling Multilevel Exploration and Sensemaking with Large Language Models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 1, 18 pages. https://doi.org/10.1145/3586183.3606756

[49] Desney Tan, Brian Meyers, and Mary Czerwinski. 2004. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *Extended Abstracts of Proceedings of ACM Human Factors in Computing Systems CHI 2004* (extended abstracts of proceedings of acm human factors in computing systems chi 2004 ed.). 1525–1528. https://www.microsoft.com/en-us/research/publication/wincuts-manipulating-arbitrary-window-regions-for-more-effective-use-of-screen-space/

[50] Kesler Tanner, Naomi Johnson, and James A. Landay. 2019. Poirot: A Web Inspector for Designers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300758

[51] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *The Journal of Open Source Software* 3, 32 (2018). https://doi.org/10.21105/joss.01057

[52] Inc. Webflow. n.d.. Webflow. https://webflow.com/ Accessed: 2025-04-10.

[53] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis* (2nd ed.). Springer Publishing Company, Incorporated.

[54] Leland Wilkinson. 2011. The grammar of graphics. In *Handbook of computational statistics: Concepts and methods*. Springer, 375–414.

[55] Inc. Wix. n.d.. Wix. https://wix.com/ Accessed: 2025-04-10.

[56] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 2648–2659. https://doi.org/10.1145/3025453.3025768

[57] Xiong Zhang and Philip J. Guo. 2018. Fusion: Opportunistic Web Prototyping with UI Mashups. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 951–962. https://doi.org/10.1145/3242587.3242632

[58] Jonathan Zong, Dhiraj Barnwal, Rupayan Neogy, and Arvind Satyanarayan. 2021. Lyra 2: Designing Interactive Visualizations by Demonstration. *IEEE Transactions on Visualization & Computer Graphics (Proc. IEEE InfoVis)* (2021). https://doi.org/10.1109/TVCG.2020.3030367

[59] Jonathan Zong, Josh Pollock, Dylan Wootton, and Arvind Satyanarayan. 2023. Animated Vega-Lite: Unifying Animation with a Grammar of Interactive Graphics. *IEEE Transactions on Visualization & Computer Graphics (Proc. IEEE VIS)* (2023). https://doi.org/10.1109/TVCG.2022.3209369