Universal Access in the Information Society
https://doi.org/10.1007/s10209-017-0606-0

LONG PAPER

A microservice-based architecture for enhancing the user experience in cross-device distributed mashup UIs with multiple forms of interaction

Antonio Jesús Fernández-García[1] · Luis Iribarne[1] · Antonio Corral[1] · Javier Criado[1] · James Z. Wang[2]

Abstract
Mobility and continuous connection entail the emergence of heterogeneous devices with multiple forms of interaction. However, it is challenging for developers and corporations to keep up with the devices and provide applications adapted to them. Besides, better user experiences attuned to users' needs and desires are increasingly in demand. User interfaces play a major role because they must be distributed through different devices and offer a customized experience for each user–device combination. We take advantage of the component-based applications easiness to build customized interfaces that can give optimal solutions to fulfill the requirements for adapting themselves to cross-device applications with multiple forms of interaction. User interaction on mashup interfaces can generate a great deal of data, which can be analyzed for improving the interaction and usefulness of the applications. In our paper, we have created a microservice-based architecture that generates datasets which contain the user behavior for further analysis. Therefore, the user experience and usability in distributed user interfaces may be improved through prediction models generated from the data. Each microservice autonomously fetches its own data and performs consistently so that it can transform datasets optimally by using feature engineering techniques. Thus, data analysis and algorithms can create accurate yet simple prediction models that provide useful knowledge to enhance the user experience. A REST API web service is added to each microservice to facilitate their communication with other microservices and/or third-party clients. The entire microservice architecture, including feature engineering and RESTful API web services for each microservice, offers an infrastructure to handle and process data interaction of cross-devices applications with multiple forms of interaction. This approach has been deployed in a real mashup application where new datasets have been created, processed and validated.

Keywords Cross-device applications · Multiform interaction · Distributed interfaces · Microservice architectures · Feature engineering · Mashup user interfaces

1 Introduction

Nowadays, it is nearly impossible to find a successful software application that only works with one type of device. Actually, mobility has become extremely important and people need to be continuously connected not only in a work environment but also in their social lives, free time and leisure activities. Desktop computers and laptops are not enough to provide such an ongoing connection and users therefore access to software applications through heterogeneous devices such as tablets, phablets, smartphones or even wearables like smartwatches and smartbands, or any other smart home devices such as lighting systems and some home automation devices. The number of heterogeneous devices

✉ Javier Criado
javi.criado@ual.es

Antonio Jesús Fernández-García
ajfernandez@ual.es

Luis Iribarne
luis.iribarne@ual.es

Antonio Corral
acorral@ual.es

James Z. Wang
jwang@ist.psu.edu

[1] Applied Computing Group, University of Almeria, Almería, Spain

[2] College of Information Sciences and Technology, The Pennsylvania State University, State College, PA, USA

Published online: 06 December 2017

that access software applications is continuously growing, and it seems that this trend continues upwards [20].

The appearance of these new devices entails developing software applications that they will not be used in a concrete device, but instead will have to be designed to be accessed from many different kinds of devices. This *cross-device development* requires design decisions and implementation effort on different layers, involving the adaptation, distribution and migration of user interfaces and data across devices [25].

User interfaces (UI) play a prominent role in that respect since they are present in multi-device environments [8]. The same UI must be distributed through different devices with multiple forms of interaction (see Fig. 1), by means of a clear customized approach to each one of them without altering the essence and functionality of the application.

The nature of these devices is diverse, and they are accessed in different environments. Considering this, the access to the software application through a specific device also implies that the user is looking for certain functionalities that may, or may not, coincide from one device to another. This is known as context awareness, that is, the ability of a device to gather information about its context, environment, location and other nearby resources to behave accordingly. For that, the device relies on its sensors and nature, i.e., the purpose for which it was designed, to determine how its processes should operate and enhance the user experience.

Furthermore, software applications can offer different forms of interaction or user interface layout to users, depending on the type of device and the circumstances in which it is accessed. For instance, a smartphone usually is handled via touch screen, but it can be also handled via voice recognition. How users interact with interfaces is called human–computer interaction (HCI). Traditionally, this interaction has been performed by means of keyboard and mouse. Nowadays, other types of actions related to natural user interfaces (NUIs) are widely used, such as touch interfaces, gestural interface, voice recognition, virtual reality, wearables or Internet of things (IoT) solutions [38].

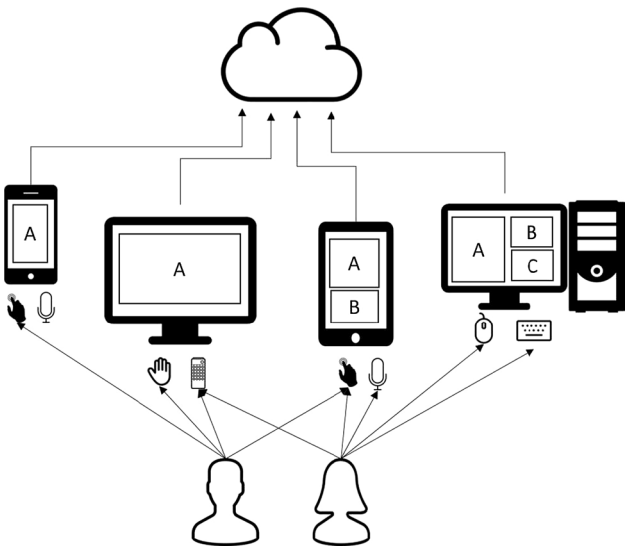

Fig. 1 Users accessing cross-device applications with multiple form of interaction

Ubiquitous computing (also called pervasive computing) [29] deals with all these heterogeneous, constantly available and connected devices where data and user interfaces are distributed. Within this concept, different types of devices from distinct locations use their computational capabilities to even determine their own behavior or to communicate with other devices everywhere. Thus, human-centered design (HCD) makes possible self-governed devices that operate autonomously without human interaction. A combination of human interaction and self-governed software applications could optimize the user experience and how users interact with devices through their UI.

This paper focuses on the aforementioned context and concepts. Users expect applications to be accessible via any device regardless of the screen size, the type of interaction or the technologies involved in it. They also expect the user interfaces to adapt to them so they can work and interact with the UI as intuitively as possible. It is therefore desired that a UI performs dynamic adaptation and evolves to the user's needs through the discovery of behavioral patterns based on the user's interaction with the interface.

To achieve that goal, we focus on mashup user interfaces [7]. Due to their granularity (coarse-grained) the mashup user interface facilitates the adaptation of their internal structure to any UI through any device. In previous works, we created a number of web services to support component-based architectures of mashup UI [9]. In these works we presented a methodology for dynamically evolving component-based architectures in order to best adapt to the user's requirements. Here, every adjustment made in the system is based on a number of adaptation rules stored in a rule repositories. Based on the users' behavior interacting with the user interface, we update the rule repositories so that the application can evolve over time.

Therefore, based on mashup UI distributed through different devices, we collect data about users' interaction. The kind of device and the form of interaction are important characteristics because they affect the way in which users handle a device. We plan to analyze the data collected to gain some knowledge about how users interact with the UI. Obtaining these data is not an easy task because the UI is distributed across many devices. We have designed an infrastructure with different microservices that work together in order to collect all the interactions performed

over distributed mashup UI (regardless of the type of device and the kind of interaction) and store them in a relational database.

This infrastructure is now extended to create a new microservice architecture that queries the relational database which contains every interaction performed by users in order to retrieve datasets. These datasets contain the users' interactions with concrete aspects of the UI, such as the components used by new users, how users customize their interfaces or which components are frequently used together, among others.

On the other hand, *feature engineering* techniques have been performed over datasets to optimize their performance, helping data analysis algorithms to create accurate and simple prediction models, which means better results [23]. Although feature engineering is usually treated lightly, it definitely plays an important role in the success of a machine learning experiment. We will go through a deep process of applying feature engineering for optimizing raw datasets. The knowledge inferred by these techniques will create new rules for self-adapting of distributed mashup user interfaces at runtime. These rules can suggest, for instance, the appropriate component that the user needs depending on the device used; redistribute the components in the UI to provide a better user experience; allow users to discover new components that can be appealing to them; or automatically optimize the components appearance according to the interaction, and so on.

Thereby, in this paper, we present a microservice architecture that creates datasets from a relational database that contains data about interactions performed on mashup UIs. Furthermore, these data can be generated from cross-device applications with distributed user interfaces in multi-device environments. The data gathered are optimized by using feature engineering techniques for further data analysis purposes.

To validate the creation process of datasets, an empirical case study is provided. An implementation has been deployed over the ENIA Project [9], a mashup user interface for environmental management used by the Andalusian environmental information network (REDIAM, Spain) [34]. The ENIA mashup user interface can be simultaneously accessible via heterogeneous devices, such as laptops, computers, smartphones and tablets by using mouse, keyboard, gesture or voice interfaces. Thus, multiple ENIA sessions can be established at the same time, by different devices at different locations. ENIA has been implemented under a software as-a-service cloud infrastructure.

The rest of the paper is organized as follows. Section 2 describes the multi-device mashup morphology and the database design for storing the interactions. Section 3 describes the microservice architecture deployed to create the interaction datasets. Section 4 explains how the architecture has been applied to ENIA and how some specific datasets have been created and optimized by using feature engineering techniques. Section 5 reviews some related projects that include cross-device developments in heterogeneous fields. Finally, Sect. 6 concludes and provides future directions.

2 Storing the mashup UI interaction

This section aims to introduce the mashup concept and illustrates its morphology through ENIA, the mashup application used in our case study. We will use the ENIA case study for that purpose because, after all, it provides a vision similar to regular mashup (being even slightly more advanced), and at the same time it introduces important particular characteristics to understand the overall work presented on this paper.

Once we have studied the ENIA mashup morphology, we will describe the operations that can be performed on the mashup UI. We must bear in mind that ENIA is a multi-device application and, as a result, some operations might not be available on some devices. For example, in smartwatch applications it is not possible to resize a component because they can only be full screen visualized on these devices. This is because the ENIA UI is distributed across multiple devices and adapted to each one of them with its peculiarities.

Likewise, an interaction could be performed differently depending on the type of interaction and device used. For example, opening a map in a laptop is usually done by using a mouse and a keyboard but with smartphones, it could be performed by using a voice recognition system (maybe because the user is driving). This will be taken into account when storing data because, although the operation is the same, the form in which it is carried out is different. We will also consider the environment in which an operation is performed, as the context awareness is a significant factor in distributed interfaces and ubiquitous computing.

Finally, we propose a relational database schema to store the interaction in ENIA. We describe the data acquisition process, and we show how the user interaction is stored in the database containing the HCI data from users across multi-devices. This information is subsequently used in the post-storage process to infer knowledge that can help to evolve the mashup UI and enhance the user experience.

2.1 Mashup UI morphology

As a definition, mashups are said to be a specific type of software that is intended to group services from different sources in the same application. Mashup user interfaces (mashup UI) integrate one or more components from one or more sources to create a unique UI that combines different components that might or might not have relationship among

them [18, 21]. Figure 2 shows a screenshot of ENIA, our mashup case study. ENIA [9] is a component-based graphical user interface for the management of environmental information *n*.

As mentioned above, ENIA has been developed for the Andalusian environmental information network [34]. Different user profiles and several categories of components exist in the ENIA interface. Examples of these user profiles include tourist, farmer or politicians. In addition, there are several categories of components in the ENIA interface. It is possible to distinguish three main types of components in ENIA, even though mashup applications are very open in that sense since they can incorporate any kind of content by placing it in a component. In ENIA, components are categorized as follows:

- OGC services. Open geospatial consortium (OGC) [27] services components have geospatial information that is placed into a map. Examples of these components include natural areas, wetlands or biosphere reserves, positioned within maps, among other kinds of OGC components.
- Social networks. These components contain social information which may be useful for users. Examples of these components include Facebook, Twitter, RSS feeds, among others.
- Applications. Other applications are registered in ENIA to be offered to users. ENIA allows third-party developers to create their own component and register it in ENIA so that users can access them. The REDIAM has a subcategory to register its own components such as beaches temperature or orthophotography images. Other examples from third-party developers are weather or clock components.

Figure 2 locates the main parts that make up the ENIA mashup UI:

(a) *Services menu* Contains the list of *Services* offered by the mashup. This menu is organized in categories and subcategories to facilitate the search of *Services*.
(b) *Services* Describe the capacities provided by the mashup UI. These capacities are offered to users and ready to be included in their workspaces. When a service is incorporated to the workspace, it is instantiated and managed as a *Component*.
(c) *Components* Correspond to *Services* which have been added to the *Workspace* and, consequently, instantiated for their use. These components are deployed inside *COTSgets*.
(d) *COTSgets* COTSgets are containers of Components. These elements have a set of attributes (e.g., width,

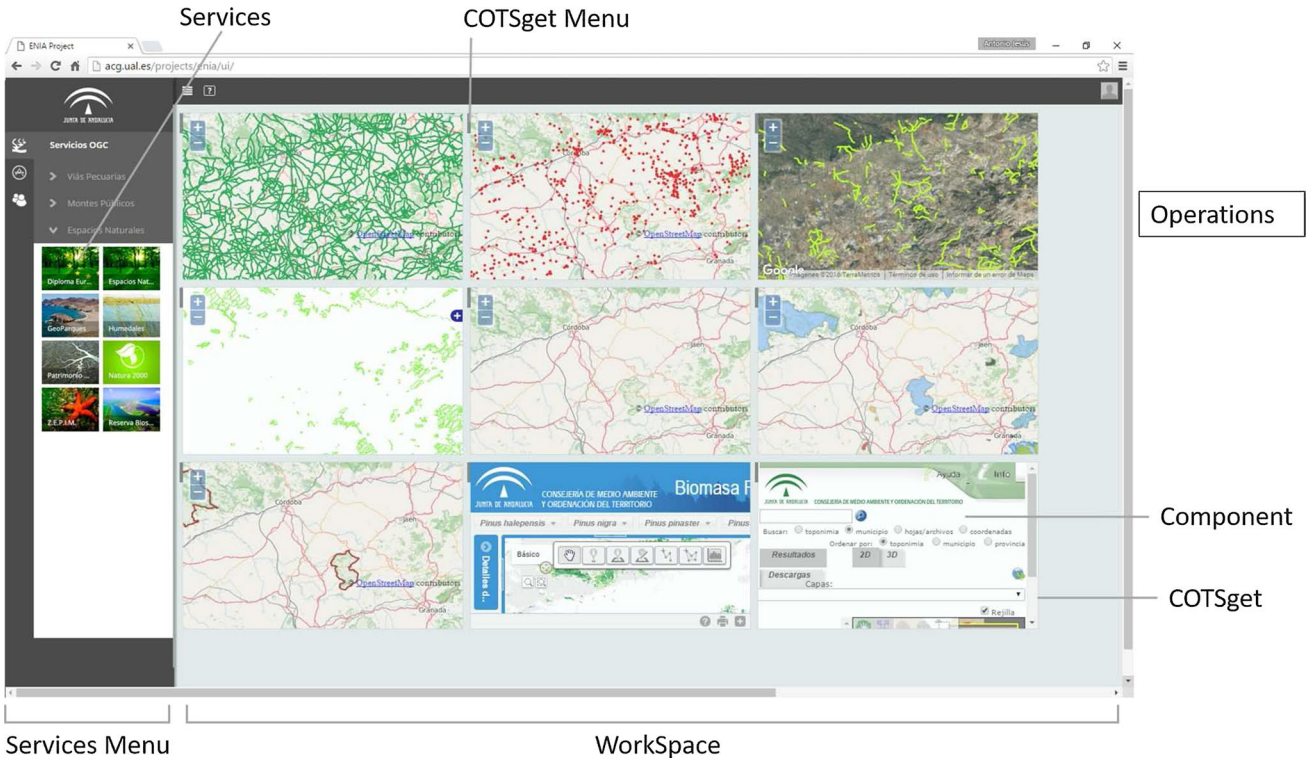

Fig. 2 ENIA mashup UI conceptual design

height or position). COTSgets are used to group *Components* with common properties, for example, components with geospatial information represented in a map. The name COTSget comes from a combination of COTS (commercial off the shelf components) and gadgets (a software element that encapsulates the functionality needed to perform a task).

(e) *COTSget menus* Provides some capabilities to interact with the *COTSgets*. Menus are toolbars placed at the top displaying some actions that can be performed over the *COTSgets*, for example, a button to remove them from the *Workspace*.

(f) *Workspace* Constitutes the work area where *COTSgets* are deployed and users can interact with them.

(g) *Operations* Are formed by a subset of actions that can be performed over the mashup UI and which are relevant for learning about the interaction.

Therefore, the ENIA mashup ($\mathcal{E}$) is summarized in the following manner $\mathcal{E} = \{\mathcal{S}, \overline{\mathcal{S}}, \overline{\mathcal{C}}, \mathcal{C}, \mathcal{W}, \mathcal{O}\}$. Thus, $\mathcal{E}$ is comprised of a set of services $\mathcal{S}$, a service menu $\overline{\mathcal{S}}$, a set of COTSgets $\overline{\mathcal{C}}$, a set of components $\mathcal{C}$, a workspace $\mathcal{W}$ and a set of operations $\mathcal{O}$. The sets of *Services* $\mathcal{S}$ are defined as $\mathcal{S} = \{S_1, S_2, \ldots, S_N$ where N is the number of *Services* registered in ENIA. The sets of COTSget $\overline{\mathcal{C}}$ are defined as $\overline{\mathcal{C}} = \{\overline{C}_1, \overline{C}_2, \ldots, \overline{C}_L\}$ where L is the number of COTSget created in the *Workspace* $\mathcal{W}$.

A concrete COTSget $\overline{C}_i$ has some properties, so it could be defined as $\overline{C}_i = \{PosX, PosY, Width, Height\}$. All the components contained in that specific COTSget share the values of *PosX*, *PosY*, *Width* and *Height* properties. Finally, the sets of operations are defined as $\mathcal{O} = \{Add, Delete, Move, ResizeBigger, ResizeSmaller, ResizeShape, Group, Ungroup, AddGroup, UngroupDelete, UngroupGroup, Maximize, Minimize\}$.

2.2 ENIA mashup UI operations

The operations that can be performed over the ENIA mashup UI are described below. A distinction is made to clear up whether the operation is standard for all mashups or specific to the ENIA mashup UI.

- `Add`. *Mashup standard operation*. Consists in adding a service to the workspace from the services menu, so it is instantiated into a component. When instantiating in ENIA, a new COTSget is created and the new component is placed into it. Some properties such as position in the *x* axis, position in the *y* axis, width and height are assigned to the COTSget.
- `AddGroup`. *Exclusively from ENIA*. Consists in adding a service to the workspace from the services menu, so it is instantiated into a component, placing it in an existing COTSget. When instantiating the component, it takes the properties previously assigned to the COTSget that contains it.
- `Group`. *Exclusively from ENIA*. Consists in placing a component that is alone in a COTSget, in another existing COTSget. The original COTSget is deleted because it is empty.
- `Ungroup`. *Exclusively from ENIA*. Consists in placing a component of an existing COTSget with more than one component, in a new COTSget that will be created specifically to that end; hence, the component will be the only one that populates the new COTSget container.
- `Delete`. *Mashup standard operation*. Consists in removing the last Component of a COTSget from the workspace. The COTSget that contains the components is also removed because it is empty now. More than one component can be deleted at the same time if they are all contained in the same COTSget and the user deletes the COTSget itself.
- `UngroupDelete`. *Exclusively from ENIA*. Consists in deleting a component from a COTSget and leaves one or more components inside it.
- `UngroupGroup`. *Exclusively from ENIA*. Consists in placing a component that is inside of a COTSget with more than one component, into another existing COTSget.
- `Resize`. *Mashup standard operation*. Consists in changing the size assigned to a COTSget and consequently, every component contained therein. It modifies the 'width,' 'height' or both properties.
- `Move`. *Mashup standard operation*. Consists in changing the position of a COTSget and therefore, to all the components that contain. It modifies the properties *PosX*, *PosY* or both of them.
- `Maximize`. *Exclusively from ENIA*. Consists in increasing the size of a COTSget to show it at full screen. It applies to all the components contained in a COTSget.
- `Minimize`. *Exclusively from ENIA*. Consists in giving back the *width, height, posX* and *posY* values to a COTSget that was previously maximized. The COTSget and all the components contained in it go back to their original position and they are no longer shown at full screen.

2.3 Database design for storing interactions in ENIA

When an interaction occurs in the mashup application, a data acquisition process is triggered in order to save all the information regarding that interaction. This information is the basis for further post-storage processes which will be explained later.

Together with the operation performed that triggers the interaction, it is convenient to save the information about the user that generates it, as well as the affected components. Since the mashup UI is distributed in different devices it is necessary to store the device that hosts the operation (laptop, smartphone, tablet, etc.) and also the form of interaction (touch, gestural, voice, etc.). It is likewise advisable to save the state that remains in the workspace after the operation. Although storing all the workspace might seem rather costly, it would make it possible to rebuild all the users' behavior step by step through the interfaces in case further analysis, not considered at design time, is required.

In order to store interaction data performed by users on the mashup UI, it is necessary to define a relational database model that would be able to store all the relevant information of the interaction. This relational database should be as complete as possible to have a good understanding of the interaction itself and the circumstances that surround it. Figure 3 shows the relational database schema that stores all the interactions performed as well as the size and position of each COTSget in the workspace after the operations are performed. Notice the presence of indexes in every table to facilitate the efficiency when querying the databases.

Each row of the `Interactions` table corresponds to an interaction performed by the user. The `operationPerformed` field saves the kind of operation performed, and the `dateTime` field saves both date and time when the interaction happened. Moreover, a wide range of fields are included to store the context information, for instance: `latitude`, `longitude`, `city`, `country`,

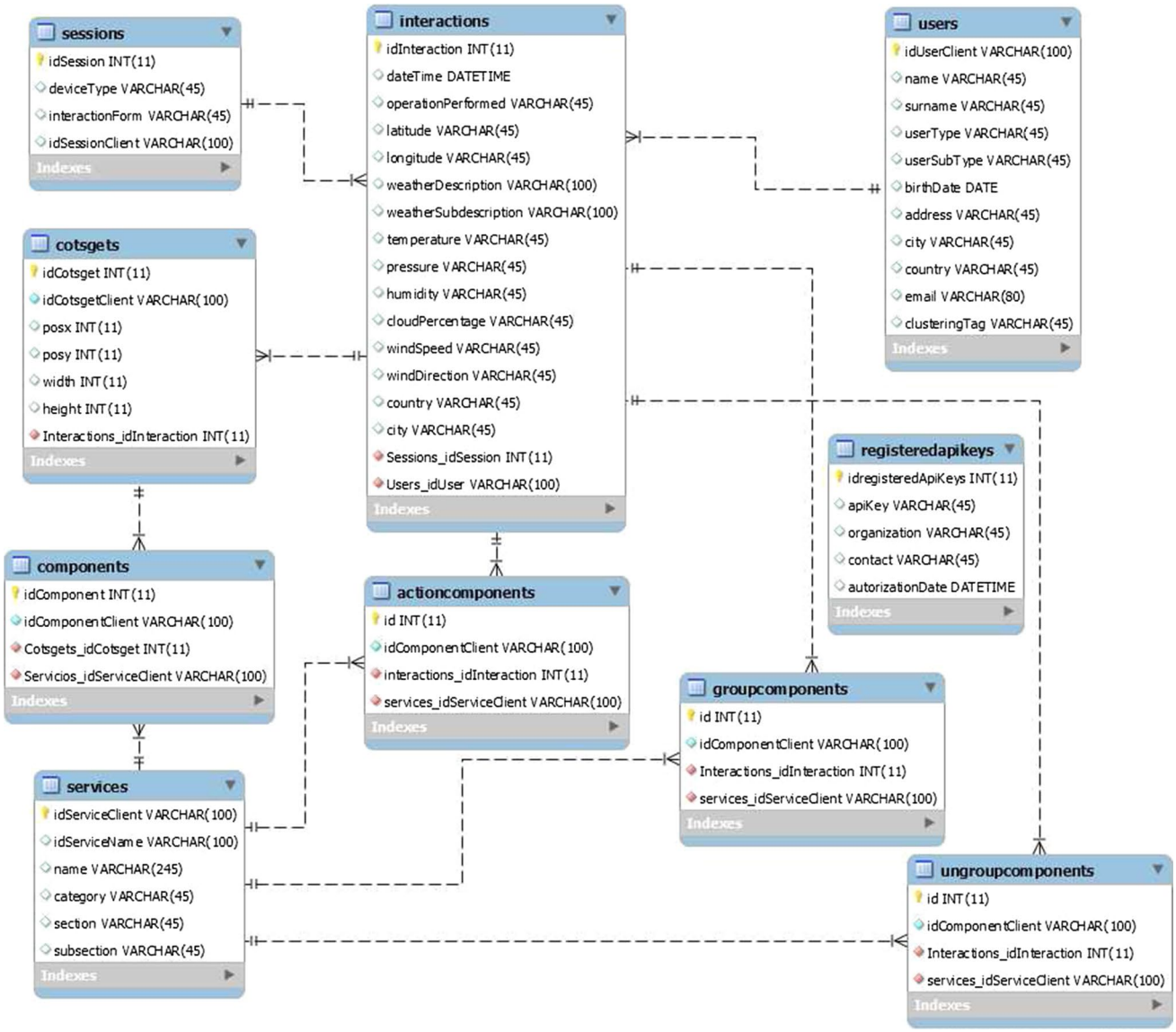

Fig. 3 Database schema to storage interaction in ENIA

weatherDescription, weatherSubdescription, temperature, pressure, humidity, cloudPercentage, windSpeed and windDirection.

The Interactions table is related to the Sessions table; thus, all the operations performed in the same session are grouped. Note that, the deviceType and the interactionForm fields are in the Sessions table. ENIA is prepared to work through different devices simultaneously with multiple forms of interaction; besides the classical mouse and the keyboard, we can add smartphones, tablets and other devices with gesture or voice form of interaction.

The Users table, which is related to the Interactions table, stores information of all users registered. In ENIA, guest users are allowed to have access the application. For that kind of users, there is a specific row in the Users table that identifies them as guests. ENIA saves extra information about Users that should enter in order to be registered in the application. That extra information is: name, surname, birthDate, address, city, country and email. The userType and userSubType fields are used to categorize users, as it has been previously discussed.

The Cotsgets table includes all COTSgets that populate the workspace after an interaction has been performed. The Components table, which is related to the Cotsgets table, stores information about all components in the workspace and indicates in which COTSget they are contained. With the information retrieved from these tables, it is possible to rebuild the workspace exactly the same as it was before the interaction was performed. The attributes posx, posy, width and height are enough to locate each COTSget in the workspace.

The Services table, which is related to the Components table, stores the information about all services registered in ENIA. The category, section and subsection fields represent the different kinds of services. That makes them easy to find, locate and use in the ENIA user interface.

It is important to identify the component on which the operation is performed because this is the component that provokes the interaction. Hosting together more than one component is not an easy task in ENIA due to the use of COTSgets. Sometimes, specific operations are not performed on just one component but on a set of components contained in a COTSget. Furthermore, components that are not included in an operation can play a second role, because they can be directly affected by the operation. For example, it happens when a component is grouped in a COTSget with more components or when a component is ungrouped from a COTSget where at least a component is left. To successfully store all that kind of interactions that happen, we make use of the ActionComponents, GroupedComponents and UngroupedComponents tables, which are all related to the Interactions table. The first one gathers the components that provoke the interaction. The second one gathers all components that had previously been in a COTSget before an Add or AddGroup operation was performed. The last one gathers the components that are left in a COTSget, when an Ungroup or DeleteUngroup operation has been performed.

There is also the registeredApiKeys table that has no relations with other tables of the database. This table is not necessary to capture the interaction, but it takes a role in the security of the stored data.

2.4 Data acquisition process

Once the relational database model is designed, it is necessary to create a process to store the interaction performed over the distributed user interfaces in the database. A data acquisition process has been created and implemented to collect the HCI interaction from the distributed mashup user interfaces and store it in the database. The stored information, in a structured form, can be exploited in future works for different purposes such as *data searching*, *data mining*, *marketing*, *security (user access and behavior analysis)*, *accessibility*, *usability* or *traceability*, among others. In our concrete case, this database will be exploited to create UIs that will evolve over time by using data mining techniques. Figure 4 shows the steps that are necessary to complete the data acquisition process.

The first step (#1) in the data acquisition process is to create a web service (called getInteraction) to receive all the interactions produced in the mashup UI. The mashup UI client is responsible for calling this web service when an interaction occurs, sending all the data that are needed to be stored. A JSON Schema is provided to the mashup UI client. This schema describes the data structure needed. Thus, clients can send JSON files and validate them against the schema, to guarantee that they are both 'well-formed' and 'valid.'

In the event that further data about a user, a service or any other aspect of the mashup UI is needed, the next step is to get to that data (#2). For that, it is commonly necessary to check the mashup application to see whether it provides web services and connect with them, if there are any. If so, the request to the web services for the data needed should be implemented in the data acquisition process.

Sometimes there is also some context information that contributes to useful data. When designing the data acquisition process is highly advisable to study the environment of the mashup UI and define which data would be worthy to obtain from the context. Usually, access to context awareness involves connecting with third-party web services. When the context data are defined and the services that provide such context awareness are identified,

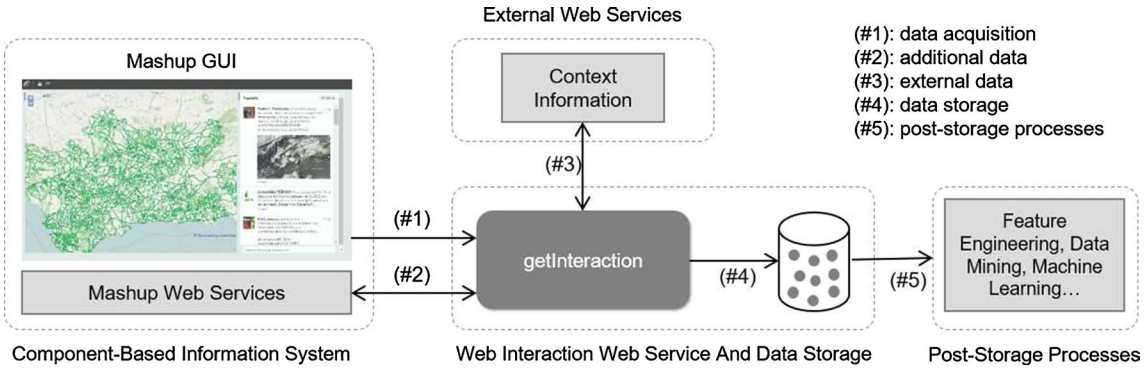

Fig. 4 Steps of the data acquisition process

the request for these external web services must be implemented in the data acquisition process (#3).

When all these steps are completed and all the required data are gathered, it should be stored in a database (#4). It is required to serialize the database operations in order to avoid integration conflicts when adding relations to the data.

Once the data are stored in a structured database, the ways to use that information are as follow (#5):

(a) *Data searching* The actions performed by users are always preserved, and they can be accessed at any time, but they are not likely to be modified.
(b) *Data mining* Discovering of patterns in large datasets created from the stored interaction data is highly advisable. Through artificial intelligence, statistics or machine learning algorithms, the user behavior can be analyzed.
(c) *Marketing* Data can be handled for marketing purposes such as promotional activities, analysis, surveys or advertising.
(d) *Security* The data stored can be secured by using role-based access control as well as it can provide users with identification for every action performed in the information system. Also, intrusion defense systems (IDS) and intrusion prevention systems (IPS) can be set up based on the storage data.
(e) *Accessibility* Through the interaction data stored, we can know how to develop an accessible design or to create assistive technology so that disable people can benefit from it.
(f) *Usability* Usability improves the design and features of the information systems to increase the effectiveness of the application when interacting with users.
(g) *Traceability* Traceability verifies all the steps performed by users when interacting with the information system. It allows chronologically reconstructing the actions performed by users.

As previously mentioned, our aim is to exploit this database to infer knowledge for creating a seamless process that could work autonomously toward the evolution of distributed multi-device mashups user interfaces by using data mining techniques.

3 Architecture of microservices and datasets

Once we have stored the interaction data, and the data acquisition process is autonomously working, feeding continuously the database with new interaction data from the mashup user interface, we will focus on processing that data.

It is worth remembering that the database schema to store the interaction was designed with the aim of losing none of the nuances of the interaction performed. With these data, it is possible to fully recreate the interactions made by users in the application step by step. For that reason, in case of using the data for a concrete purpose, it is necessary to query the relational database and retrieve the data we need for a specific goal.

Given that there are many possibilities to exploit the stored data, which may be complementary or not, a microservice-based architecture is proposed. The microservice-based architecture structures the application as a modular set of services that collaborate together avoiding the monolithic applications difficulty of decomposing or scaling. In this architecture, microservices [12, 24, 35] are independently deployable services where each component in the system is a stand-alone entity that interacts with others across a network with a well-defined interface. Each microservice added to the pool has a concrete purpose, significantly different from others.

We are interested in analyzing the user's behavior and for that reason; we want to obtain the interaction data for addressing the creation of datasets, each dataset focusing on a specific part of the HCI that we want to enhance. For

example, if we want to facilitate the use of the application to new users, the system can suggest them components when they are newly registered in the information system. To do this, it makes sense to retrieve from the database the first interactions performed from previous users (especially users that have something in common) and create a dataset that contains that data.

This architecture of microservices will allow developers (or groups of developers) creating datasets to work in parallel at any time. This architecture also facilitates continuous integration (CI) and continuous delivery (CD), as it allows us to produce software in short cycles. This will ensure that the software can be reliably released at any time [3, 19]. The whole pool of microservices has defined boundaries and complies with the interface segregation principle (ISP), one of the SOLID [2, 13] principles, which states that clients should not be forced to depend on methods that they do not use. This is really useful when it is necessary to refactor, change or redeploy the system. Also, due to the microservice-based architecture granularity, this architecture is well oriented to distributed approaches. The coarse- and medium-grained granularity of components in mashup applications enables us to manipulate components and easily adapt them to multi-device distributed user interfaces.

Furthermore, this architecture allows us to have concrete microservices to serve one specific purpose coded with that focus. This is highly aligned with the single responsibility principle (SRP) from SOLID, which says that every module should be responsible for a single part of the functionality, which should be entirely encapsulated by the class. If one microservice fails, the others will continue working properly because the functionality of each microservice is isolated.

Additionally, each microservice may have their own web service; thus, they can expose their functionality to others services or third-party clients, if allowed. In the case we pursue, they can be accessed through their web service (further explained) for creating datasets, which could be the input for machine learning algorithms or ways of data analysis. Establishing a connection with the work presented on this paper, some of the purposes we are interested to cover that can be encapsulated in microservices can be (a) discover the components most commonly used by new users, (b) analyze how the use of a particular device impacts on the usage of a component or (c) suggest the usage of some specific components depending on the form of interaction employed to manipulate the device, among others examples.

Figure 5 shows the datasets creation process, through a microservice-based architecture. There is a pool of microservices, in which each microservice can connect to the database through the database controller in order to obtain the data that it needs for its purpose (which is the first part of the process, the data acquisition).

Usually, the acquired data are not optimized to be analyzed; therefore, a feature engineering process is applied, whereby the datasets are optimized to be the best possible input for applying the algorithm that is intended in each specific case.

4 ENIA, microservices and datasets

Once the microservice-based architecture has been theoretically described and its suitability to create datasets has been discussed, we are going to explain now how this architecture

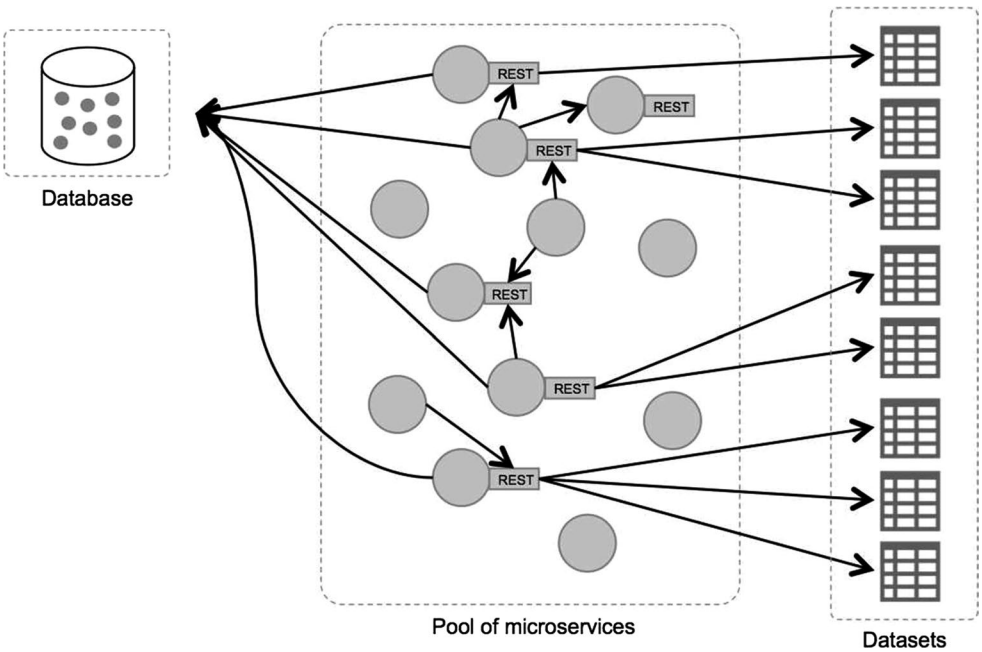

Fig. 5 Microservices-based architecture for addressing the creation of datasets

has been deployed in ENIA. In order to illustrate that, we are going to create a microservice and add it to the pool of microservices.

The ENIA user interface case study is an implementation already deployed [9] where approximately a dozen users from the REDIAM (Andalusian environmental information network) are continuously using the application. Furthermore, the application is not only used for these users, but it is an open framework available for any citizen with different profile of using in the system. The main kinds of users (profiles) defined in ENIA are the following:

(a) *Rediam staff* In their daily work they internally exploit information contained in ENIA components for different purposes.
(b) *Farmers* Use components for agriculture purposes such as looking up types of soil, weather forecast or locating wetlands.
(c) *Politicians* Use components to analyze data for different purposes according to their needs.
(d) *Tourists* Use components for touristic purposes such as finding touristic spots or to know the water temperature in beaches.
(e) *Guests* Users that are not registered in ENIA can access and use it freely.

In this section, we are going to deeply describe how a microservice works in the architecture, as well as describe each one of the processes that have to be attended when deploying a microservice into production. In this case, microservices are going to be built to generate datasets. These datasets can be used for multiple purposes such as data analysis, reports or logs. The main processes of these microservices are:

- Obtain the interaction data needed by querying the relational database.
- Apply feature engineering to optimize the dataset for further analysis.
- Create a web service that enables the microservice to communicate with other microservices or entities. It allows us to potentially scale the system functionality by improving the collaboration between parts in a distributed and ubiquitous system.
- Automatize the execution of the microservice (if it is required).
- Enable an interface to receive feedback from other services (it can be integrated into the web service).

In the second part of this section, the feature engineering process will be deeply explained. We will cover many aspects of great importance in the process with the aim of optimizing the datasets and getting better results when using machine learning algorithms. Some of these aspects are cleaning data, discretization of features or splitting instances, among others.

To illustrate the process, through this section we will focus on a concrete microservice that enhances the user experience. The goal of the new microservice will be the suggestion of useful components to new registered user when they interact with the mashup. By doing that, we want to increase the number of *add* and *addGroup* operations that a user carry out and he/she may find it interesting when interacting with the mashup user interface through heterogeneous devices with multiple forms of interaction. Thus, the user's engagement with the user interface (UI) will increase, enhancing the user experience.

4.1 Deploying a new microservice in the architecture

The steps taken to deploy a new microservice in the microservice-based architecture are shown in Fig. 6 and described in this subsection.

First of all, a dataset has to be obtained from the original source. It can be done by querying the origin database or by acquiring the dataset through a web service or data files (CSV, XLS...). Usually, this dataset has too raw data, and feature engineering techniques are applied to optimize it. A REST web service may be included in each microservice of the system to communicate and send data to the machine learning algorithms. In addition, this REST service can be also useful to provide access to third-party applications, and to facilitate the communication with other available microservices, or even to receive some feedback from the machine learning algorithms.

After that, the methodology proposes the creation of an automatized process that periodically generates accessible and updated datasets without the need for generating them in real time. Finally, feedback from the machine learning algorithms can be obtained and saved through the REST web service. Further details about the steps of the process are explained in the next subsections (for that, each step of the process will be explained in a separate subsection).

4.1.1 Acquiring data

The first step of the process is to acquire from the relational database the user's interaction data to build the dataset. To explain this issue, let us suppose the following target sentence that will be used through the paper as a running example and starting point:

"*Suggest components to new registered users*"

This previous sentence would require the following type of data from the database:

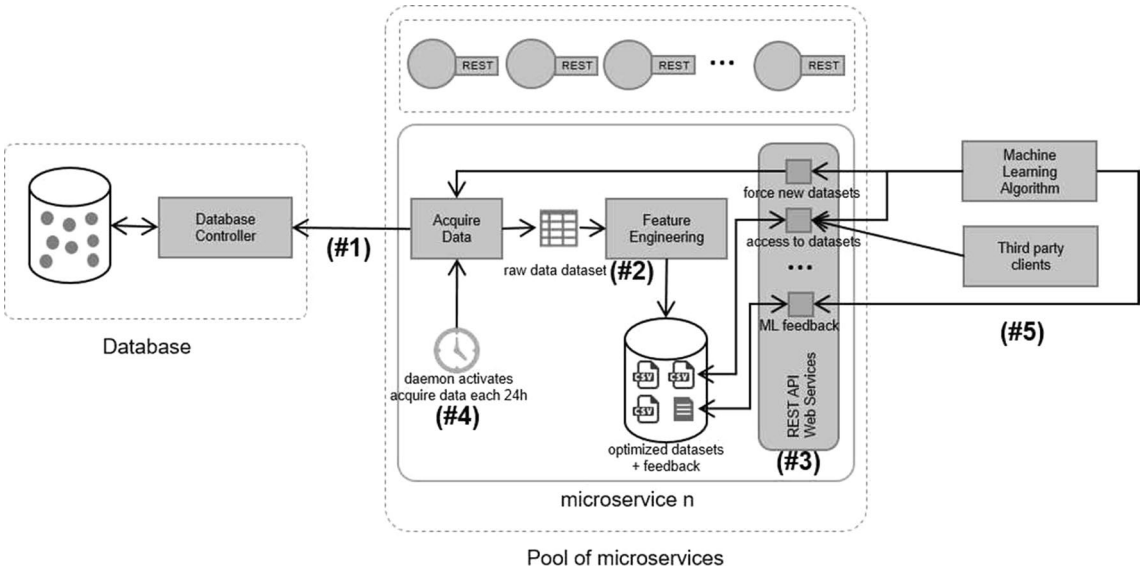

Fig. 6 Process inside a microservice

- *Interaction data type*. The `idInteraction`, `operationPerformed` and `actionComponents` fields of the `interactions` table contain the operation performed as well as the components over which the interaction has affected. Note that `actionComponents` is an array that contains one or more values because an operation can be performed in COTSgets that could contain more than one component at the same time.
- *Cross-device, form of interaction and simultaneous sessions*. Data about the simultaneous sessions in which the interaction has been performed (which include the device type and the form of interaction used) can be retrieved from the `idSession`, `deviceType` and `interactionForm` fields in the `sessions` table (see Fig. 3).
- *Data about the user that performs the operation*. The user information and the user profile (for instance, a farmer, a politician, a tourist, among others) can be accessed from the fields `idUserClient`, `birthDate`, `country`, `userType` and `userSubType` in the `users` table (Fig. 3).
- *Data about the context awareness*. Data such as date, time, location or weather can be retrieved from the `dateTime`, `latitude`, `longitude`, `temperature` and `weatherDescription` fields in the `interactions` table (see Fig. 3).

Therefore, it is meaningful to acquire the following information in terms of a dataset:

```
Dataset = {
  idInteration,
  operationPerformed,
  actionComponents[],
  idSesstion,
  deviceType,
  interactionForm,
  idUserClient,
  birthDate,
  country,
  userType,
  userSubType,
  dateTime,
  latitude,
  longitude,
  temperature,
  weatherDescription
}
```

In order to facilitate the explanation of further steps the dataset will be summarized in the next fields:

```
Dataset = {
  idUserClient,
  userType,
  idSession,
  idInteraction,
  operationPerformed,
  deviceType,
  interactionForm,
  dateTime,
  latitude,
  longitude,
  actionComponents[]
}
```

The dataset with these fields can be seen in Table 1, included at the end of the paper. It contains the necessary data about the interactions and the components affected; the sessions, devices and forms of interaction related to the mashup UI in which the interactions have been performed; information about the users that performs such interactions; as well as information about the context surrounding each interaction.

4.1.2 Feature engineering

Usually, data are too raw for learning and the dataset retrieved directly from the database must be processed before applying data analysis. *Feature engineering* is a process that transforms these raw data to create features that have better representation and thus be able to create better predictive models [30]. The quality of the predictions models created with machine learning algorithms depends on the feature engineering approach that has been followed where usually better features mean better prediction models.

Feature engineering gets the most of the data available by creating pieces of information (features), which might be useful for prediction. The quantity and quality of the features can define how good the model obtained by the algorithm is. Since algorithms are pretty standard, it is worth spending time doing a good feature engineering work to get good results. To successfully perform feature engineering over data, a domain understanding of the problem as well as a deep algorithm understanding is needed.

This is an important issue to be considered, and later in this section we will explain in depth how feature engineering is applied to the case study dataset. For that, we will transform the dataset shown in Table 1 and optimize it to increase the success of machine learning algorithms.

4.1.3 Microservice REST web services

As previously commented, microservices have to expose their functionality to others. In our case, every microservice developed to create datasets, at least, has to provide access to the dataset when generated. Moreover, on many occasions, there are communications between microservices and we have to provide the way for this to happen. For that reason, some microservices implement a REST web service.

We decided to use REST against SOAP [33] protocol because it is more prevalent in the industry due to its flexibility and simplicity [39]. REST has a better performance and scalability, in general, and SOAP requires more bandwidth and resources than REST. Besides, REST permits different data formats such as plain-text, JSON, XML, HTML, among others, and SOAP permits XML data format only.

In addition, the microservices could expose its functionality via a REST API to be used by third-party applications. There could be a problem with this because typically, microservices provide fine-grained APIs. It implies that clients who need to interact with several datasets may find it a bit tricky to get to know how to access each of them. Consequently, a gateway is proposed as a solution. The gateway will be the single entry point for all clients when accessing to the microservices architecture; we prevent them from knowing what everyone offers. They just directly access to the API gateway that canalizes the architecture web services to third-party entities. We do not strictly need this because we can access directly to the REST API of the microservices (although it may be useful).

In our case study, datasets can directly be accessed through a URL. Additionally, we decided to implement a web service that can request in real time a new dataset. If the time generating the dataset is considerable, the client can be notified when the dataset is ready.

4.1.4 Automating the process of creating datasets

Datasets can be generated on demand by other microservices or by third-party clients accessing the REST API. Usually, it is not necessary to generate the datasets at the moment of the request, in real time, due to mainly two reasons. First, the generation of a dataset can consume a long time that could provoke a delay in the answer. Second, usually, accessing to datasets from a few hours ago or even a few days ago poses no appreciable inconvenience when treating with this kind of data, unless otherwise indicated.

In ENIA, to automatize this process, a daemon is periodically requested to call the APIs of each microservice and generates updated datasets. By default, in ENIA, each microservice generates the datasets once a day. Datasets can be generated in many formats such as HTML, JSON, XML, CSV, Excel, HTML or whatever format may be appropriate. In our implementation, when the datasets are being generated, we create a CSV file that contains all the data. These CSV datasets are stored in the cloud, and they are accessible to whoever wants to make use of them via the microservice

Table 1 Dataset before feature engineering

Inst.	#1 idUser	#2 userType	#3 idSess	#4 idInter	#5 operationPerf	#6 deviceType	#7 interForm	#8 dateTime	#9 Latitude	#10 Longit	#11 Weather	#12 Objective field actionComponents
Features												
#1	432	Farmer	2131	574161	Add	Laptop	Mouse	5/5/2016 7:44:53	40.793862	− 77.86788	Clouds	CuttleRoads
#2	432	Farmer	2131	574201	Delete	Laptop	Mouse	5/5/2016 7:44:59	40.793862	− 77.86788	Clouds	Twitter
#3	432	Farmer	2131	574221	ResizeBigger	Laptop	Mouse	5/5/2016 7:45:03	40.793862	ES.8235		CuttleRoads
#4	432	Farmer	2131	574361	AddGroup	Laptop	Mouse	5/5/2016 7:45:12	40.793862	− 77.86788	Clouds	BiosphereReserves
#5	511	Politician	2811	574601	Add	Smartphone	Touch	6/5/2016 12:10:03	37.394211	− 5.985901	Clear	WordHeritage
#6	511	Politician	2811	574621	Resize	Smartphone	Touch	6/5/2016 12:10:08	37.394211	− 5.985901	Clear	WordHeritage
#7	511	Politician	2811	574601	AddGroup	Smartphone	Touch	6/5/2016 12:10:15	37.394211	− 5.985901	Clear	GeoParks
#8	511	Politician	2811	575821	ungroupDelete	Smartphone	Touch	6/5/2016 12:20:53				WordHeritage
#9	17	Tourist	4328	577991	Add	Tablet	Voice	6/5/2016 16:22:36	36.837067	− 2.436463	Sunny	BeachTemperatures
#10	17	Tourist	4328	578221	Add	Tablet	Voice	6/5/2016 16:22:46	36.837067	− 2.436463	Sunny	Weather
#11	17	Tourist	4328	578281	Resize	Tablet	Voice	6/5/2016 16:24:08	36.837067	− 2.436463	Sunny	BeachTemperatures
#12	17	Tourist	4328	579621	Add	Tablet	Voice	6/5/2016 16:31:32	36.837067	− 2.436463	Sunny	Facebook
#13	120	Farmer	5876	580051	AddGroup	Laptop	Mouse	7/5/2016 6:51:53	40.793862	− 77.86788	Rain	Wetlands
#14	120	Farmer	5876	580071	ResizeShape	Laptop	Mouse	7/5/2016 6:52:12	40.793862	− 77.86788	Rain	CuttleRoads, Wetlands
#15	120	Farmer	5876	580121	Add	Laptop	Mouse	7/5/2016 7:02:07	40.793862	− 77.86788	Rain	Weather

REST API or even directly using a URL. When accessing through the web service, it returns directly the CSV file. When accessing through a URL, clients directly access a public repository that contains public datasets. We find it appropriate to use CSV files because they are widely used for these cases and all major machine learning and data analysis solutions manage them easily. Certainly, besides the auto-generated datasets, it is possible to request the creation of updated datasets in real time on demand, if needed, through the microservices web services.

4.1.5 Feedback from algorithms

Occasionally, microservices can receive feedback from the algorithms or the data analysis process when executed. That can happen for many reasons, one of them could be that some inferred knowledge wanted to be stored from the microservices in order to offer it to other microservices or third-party applications that can make use of it. Although it is not the purpose of this paper to talk deeply about concrete machine learning algorithms (it will be a further research), it will be useful to illustrate a situation in which this can happen.

Consider a microservice that creates datasets to be the input of a clustering machine learning algorithm. Cluster analysis consists in given a collection of unlabeled data, finding groups of them with a meaningful homogeneous structure and grouping them into clusters. If a dataset served by a microservice contains information of all ENIA users, the output of the algorithm will likely present clusters where the ENIA users are grouped according to some features that capture similarities between them. In that case, the microservice could be interested in incorporating a web service in its API to expose this knowledge to other microservices or third-party clients. The new web service, given a user, can return the cluster where it has been categorized.

4.2 Applying feature engineering

As mentioned above, feature engineering can effectively optimize the datasets and it plays a key role in machine algorithms success. To do it properly, it is crucial to study the data domain and the algorithms that process the dataset in order to work optimally with it. By knowing the data domain, some significant changes can be done in the features that will improve the overall performance of the machine learning algorithms that will process the data to create predictive models.

In this subsection, the applied feature engineering techniques are explained. We will make use of the raw dataset previously acquired from the database to apply the feature engineering process. The microservice of the case study had the purpose of creating a dataset that contains the user interaction with the mashup application UIs that are distributed across multiple devices with multiples form of interaction. The dataset will be optimized in order to be applied in machine learning algorithms; thus, a predictive model for suggesting components to new users can be built. By suggesting components to new registered users, we expect that the user experience when interacting with the mashup UIs will improve. We also expect that these suggestions of new components can be customized to the user profile and given the device he/she is using.

The raw dataset directly acquired from the relational database is:

```
Dataset = {
  idUserClient,
  userType,
  idSession,
  idInteraction,
  operationPerformed,
  deviceType,
  interactionForm,
  dateTime,
  latitude,
  longitude,
  actionComponents[]
}
```

This dataset has thousands of instances that contain the HCI with the mashup client application from hundreds of users, most of them registered in the application and classified in categories. In these instances, the interaction performed in the mashup UIs via multiple devices and forms of interaction is gathered.

A representation of the dataset with some real instances retrieved from ENIA is shown in Table 1. Each column represents a feature, i.e., a piece of information that might be useful for prediction. Each row of the table represents an instance, i.e., a record in form of a vector that contains values for each feature. The number of instances and features is simplified in order to easily illustrate the feature engineering process.

The last column of the table has a special feature usually referred as *objectiveField*. In supervised machine learning algorithms, this field is the one that wants to be predicted. In unsupervised machine learning algorithms, the *objectiveField* does not exist. Through different tables, we will easily see how feature engineering steps transform the dataset.

The parts that compound the feature engineering process implemented in this approach are as follows: (a) cleaning data to avoid errors and empty values; (b) discretizing features; (c) creating new features by splitting existing ones; (d) creating new features by merging existing ones; (e) splitting instances; (f) filtering instances; (g) adding features from algorithms feedback; (h) splitting the dataset.

Figure 7 illustrates the feature engineering pipeline that we have applied. In addition, the detailed process is explained in the next subsection.

The data in Table 1 (at the end of the paper) contain real user interactions. As commented, we want to suggest components to new registered users. For that reason, *actionComponents* will be the objective field.

4.2.1 Cleaning data

Data may contain errors. Some errors are easy to detect, and they will probably come up by themselves sooner or later. Examples of this kind of error could be that text values appear where numeric values were expected or, in general, that the format does not match with the required one. On the other hand, some errors are very difficult to find and context information is needed in order to find them. Imagine we have been reading data from a sensor which has not been working properly for a few hours. The received values are as expected, but they cannot be trusted. It is necessary to check if the values read by sensors are consistent.

Table 1 has an instance containing an error. In the instance #3, the feature #10 (`longitude`) contains a value that is not as it is expected. It contains the value `ES.8235`, which clearly does not have the format of a coordinate. A decision must be made, remove the value of this field or delete the entire instance.

Data also may contain empty values. This problem is easy to detect since features do not have any data at all. The instance #8 of Table 1 contains three empty features: #9 (`latitude`), #10 (`longitude`) and #11 (`weather`). This may have happened because the smartphone where the interaction was performed could not access to the location at that precise moment or it was not allowed by the user. The weather is obtained from a third-party web service that receives as inputs the latitude and longitude. As these two variables, `latitude` and `longitude`, could not be provided, the weather feature is empty. Note that feature #11 (`weather`) in instance #3 is also empty, probably because this instance contained an error in the #10 (`longitude`) field, and that made it impossible to obtain the weather from the third-party web service.

Some algorithms deal with empty values (so it is important to know in advance the algorithm that will be used later to analyze the dataset). If the algorithm that will be used does not manage empty values, it is possible either omit the instances that contain empty values or estimate them, if possible. In our case, we have decided to omit them, since we consider that we have enough instances for data analysis and we can afford to delete them. Table 2 shows how the dataset looks like after cleaning data.

4.2.2 Discretization of continuous features

Discretization allows constructing a meaningful range for a continuous feature according to the domain. The resulting features have better semantics, and the algorithms get out more meaningful information from them.

Feature #8 (`dateTime`) in Table 2 is a good example of this situation. Instead of raw data like `6/5/2016 12:10:03`, it is useful to transform it to a new discrete feature like `partOfTheDay` with `morning`, `afternoon`, `evening`, `night` as possible values. As we are familiar with the domain, we already assume with a high degree of certainty that, for analyzing the behavior of users like farmers or tourists, the new proposed feature has better semantics than the feature as it was before. In the same way, the same Feature #8 (`dateTime`) can also be discretized in a new feature called `season` with `fall`, `winter`, `spring`, `summer` as possible values.

We know beforehand that a tourist will more likely visit the beach on summer afternoons than on winter nights. We also know beforehand, that a farmer will more likely pick up oranges on winter mornings than on summer afternoons. Thus, we know that by transforming features as suggested, we are bound to obtain better results than with the raw `dateTime` feature.

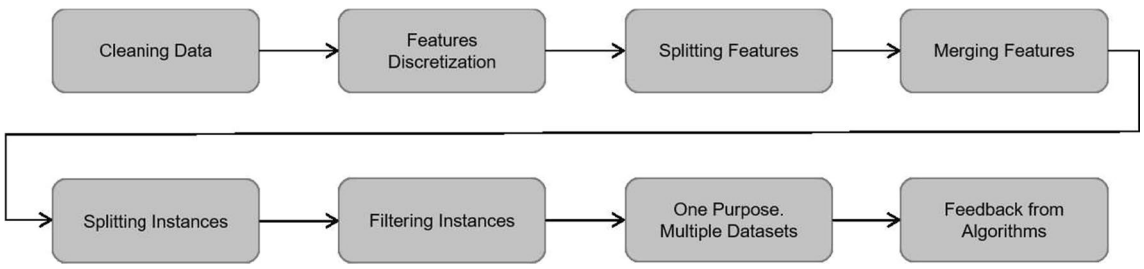

Fig. 7 Feature engineering pipeline

Table 2 Dataset after cleaning feature engineering

Inst.	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12 objective field
	idUser	userType	idSess	idInter	operationPerf	deviceType	interForm	dateTime	Latitude	Longit	Weather	actionComponents
Features												
#1	432	Farmer	2131	574161	Add	Laptop	Mouse	5/5/2016 7:44:53	40.793862	− 77.86788	Clouds	CuttleRoads
#2	432	Farmer	2131	574201	Delete	Laptop	Mouse	5/5/2016 7:44:59	40.793862	− 77.86788	Clouds	Twitter
#4	432	Farmer	2131	574361	AddGroup	Laptop	Mouse	5/5/2016 7:45:12	40.793862	− 77.86788	Clouds	BiosphereReserves
#5	511	Politician	2811	574601	Add	Smartphone	Touch	6/5/2016 12:10:03	37.394211	− 5.985901	Clear	WordHeritage
#6	511	Politician	2811	574621	Resize	Smartphone	Touch	6/5/2016 12:10:08	37.394211	− 5.985901	Clear	WordHeritage
#7	511	Politician	2811	574601	AddGroup	Smartphone	Touch	6/5/2016 12:10:15	37.394211	− 5.985901	Clear	GeoParks
#9	17	Tourist	4328	577991	Add	Tablet	Voice	6/5/2016 16:22:36	36.837067	− 2.436463	Sunny	BeachTemperatures
#10	17	Tourist	4328	578221	Add	Tablet	Voice	6/5/2016 16:22:46	36.837067	− 2.436463	Sunny	Weather
#11	17	Tourist	4328	578281	Resize	Tablet	Voice	6/5/2016 16:24:08	36.837067	− 2.436463	Sunny	BeachTemperatures
#12	17	Tourist	4328	579621	Add	Tablet	Voice	6/5/2016 16:31:32	36.837067	− 2.436463	Sunny	Facebook
#13	120	Farmer	5876	580051	AddGroup	Laptop	Mouse	7/5/2016 6:51:53	40.793862	− 77.86788	Rain	Wetlands
#14	120	Farmer	5876	580071	ResizeShape	Laptop	Mouse	7/5/2016 6:52:12	40.793862	− 77.86788	Rain	CuttleRoads, Wetlands
#15	120	Farmer	5876	580121	Add	Laptop	Mouse	7/5/2016 7:02:07	40.793862	− 77.86788	Rain	Weather

4.2.3 Creation of new features by splitting

Sometimes there are features rich enough to be transformed in more than one feature, helping us to create better prediction models. Sometimes there are features that, for a specific domain, contain great information. In these cases, a feature could be split in more than one, creating better prediction models.

Feature #8 (`dateTime`) in Table 2 is a good example of this situation, as it has hinted previously. The feature contains information about date and time. With the time information, we can create the new feature `partOfTheDay` with `morning`, `afternoon`, `evening`, `night` as possible values; similarly, with information about date, we can create the new feature `season` with `fall`, `winter`, `spring`, `summer` as possible values.

4.2.4 Creation of new features by merging

Like the previous case, there are some features that just by themselves do not contribute much to add value. However, by merging them with others features, it is possible to create meaningful features that help to the global performance of the algorithms.

In this scenario, we are looking to find components to suggest to new users in the mashup application. For that reason, it is interesting to know in our dataset if the user that has performed the interaction is a new user in the system, or not. Feature #3 (`idSession`) and feature #4 (`idInteraction`) in Table 2 provide information about the session and interaction of users, but it is too raw. By analyzing and merging these features it is possible to create a new feature `isNewUser`. This feature can also be discretized to be a boolean feature with `{yes, no}` as possible values. To determine whether a user is new or not, it is necessary to have knowledge of the domain and decide the number of sessions and/or interactions needed to demarcate the threshold between new users and old users.

There are other features that make sense to merge. Features #9 (`latitude`) and #10 (`longitude`) can be merged and discretized to a new feature: `country`. This new feature has as possible values a finite list with the names of all countries in the world. It contributes to the dataset with valuable information because users may behave differently according to their country of origin.

The new dataset after applying 'the discretization of continuous features,' 'the creation of features by splitting the existing ones' and 'the creation of new features by merging the existing' tasks is shown in Table 3 (at the end of the paper).

Table 3 Dataset after discretize, split and merge feature engineering

Inst.	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11 objective field
	idUser	userType	isNewUser	operationPerf	deviceType	interForm	partOfTheDay	Season	Country	Weather	actionComponents
Features											
#1	432	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Clouds	CuttleRoads
#2	432	Farmer	No	Delete	Laptop	Mouse	Morning	Spring	United States	Clouds	Twitter
#4	432	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Clouds	BiosphereReserves
#5	511	Politician	Yes	Add	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#6	511	Politician	Yes	Resize	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#7	511	Politician	Yes	AddGroup	Smartphone	Touch	Afternoon	Spring	Spain	Clear	GeoParks
#9	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#10	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Weather
#11	17	Tourist	Yes	Resize	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#12	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Facebook
#13	120	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Rain	Wetlands
#14	120	Farmer	No	ResizeShape	Laptop	Mouse	Morning	Spring	United States	Rain	CuttleRoads, Wetlands
#15	120	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Rain	Weather

4.2.5 Splitting instances

It can occur that certain values of features in some instances, as retrieved from the database, can contain multiple values. Even though they are correct and can be analyzed, it is advisable to separate them in several instances, due to its important contribution.

The data type of feature #4 (`actionComponents`) of Table 3 is an array. It can contain more than one value, in fact, instance #14 of that table contains the value `CuttleRoads, Wetlands`. This instance should be split into two parts. Both instances will contain the same data except for the feature #4 (`actionComponents`) field. One of them will contain the value `CuttleRoads` and the other one the value `Wetlands`. The new dataset after having applied 'the split of instances' task can be seen in Table 4, included at the end of the paper.

4.2.6 Filtering instances

There are some occasions in which all the instances that are retrieved from the database are not required for the problem which is being addressed. For that reason, some instances have to be deleted according to the values of one or more features.

Our goal is to suggest components to add to users. Therefore, looking at Table 4 the only values allowed for the feature #4 (`operationPerformed`) should be `add` or `addGroup`. In the following instances, the operation performed is not `add` or `addGroup` and they must be deleted: #2, #6, #11, #14A and #14B. The new dataset after applying the *filtering of instances* task can be seen in Table 5.

4.2.7 One purpose, multiple datasets

There are some scenarios where it makes sense having a microservice that creates more than one dataset. It can often happen because the datasets can be decomposed by applying some filters.

In this way, datasets can be decomposed into multiple datasets that have the same features, but the number of instances is filtered according to some values. Hence, taking into account the domain, it is possible to send several datasets to the same algorithm, each one aiming to obtain a specific knowledge.

In ENIA, users are categorized in profiles. At first, users categorized in the same profile should have some similarities. Therefore, it makes sense the suggestion of new components can rely not only on interactions from all users, but those performed by users from the same category, and why

Table 4 Datasets after splitting and filtering instances feature engineering

Inst.	#1 idUser	#2 userType	#3 isNewUser	#4 operationPerf	#5 deviceType	#6 interForm	#7 partOfTheDay	#8 Season	#9 Country	#10 Weather	#11 objective field actionComponents
Features											
#1	432	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Clouds	CuttleRoads
#2	432	Farmer	No	Delete	Laptop	Mouse	Morning	Spring	United States	Clouds	Twitter
#4	432	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Clouds	BiosphereReserves
#5	511	Politician	Yes	Add	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#6	511	Politician	Yes	Resize	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#7	511	Politician	Yes	AddGroup	Smartphone	Touch	Afternoon	Spring	Spain	Clear	GeoParks
#9	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#10	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Weather
#11	17	Tourist	Yes	Resize	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#12	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Facebook
#13	120	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Rain	Wetlands
#14A	120	Farmer	No	ResizeShape	Laptop	Mouse	Morning	Spring	United States	Rain	CuttleRoads
#14B	120	Farmer	No	ResizeShape	Laptop	Mouse	Morning	Spring	United States	Rain	Wetlands
#15	120	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Rain	Weather

Table 5 Dataset after filter instances feature engineering

Inst.	#1 idUser	#2 userType	#3 isNewUser	#4 operationPerf	#5 deviceType	#6 interForm	#7 partOfTheDay	#8 Season	#9 Country	#10 Weather	#11 objective field actionComponents
Features											
#1	432	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Clouds	CuttleRoads
#4	432	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Clouds	BiosphereReserves
#5	511	Politician	Yes	Add	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#7	511	Politician	Yes	AddGroup	Smartphone	Touch	Afternoon	Spring	Spain	Clear	GeoParks
#9	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#10	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Weather
#12	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Facebook
#13	120	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Rain	Wetlands
#15	120	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Rain	Weather

not, even those performed by the user in previous interactions. Some cases can be found:

- Analysis of components added by all users. One dataset will be obtained with information about all users. This is the dataset we already have and can be seen in Table 5 (at the end of the paper).
- Analysis of components added by users with the same category. *N* datasets will be obtained where *N* is the number of profiles that exists in ENIA. The knowledge inferred by the machine learning algorithm when analyzing a dataset will only be valid for users from which the profile is analyzed. Table 6 shows the datasets created for each user profile. In this situation, feature #2 (`userType`) must be deleted because it is a 'constant' in each dataset; Nevertheless, this feature has been presented in the table for clarity.
- Analysis of components added by a specific user. *M* datasets will be obtained where *M* is the number of users that exists in ENIA system. The knowledge inferred by the machine learning algorithm when analyzing a dataset will be valid only for the user analyzed. Table 7 shows the datasets created for each specific user. Note that in this situation the feature #1 (`idUser`) must be deleted because it is a constant in each dataset, although, for clarity, is still present in the table.

Datasets have been divided according to the users category and concrete users, but they could also be divided by means of the feature #5 (`deviceType`) in case we want to suggest concrete components for specific devices, or by means of the feature #6 (`interactionForm`) if we want to suggest concrete components for a specific form of interaction.

4.2.8 Feedback from algorithms

As previously discussed in this section, there could be some occasions where microservices could be interested in receiving feedback from the machine learning algorithms. Although it is not the purpose of this paper to talk about machine learning algorithms in depth, to understand this, it helps to know that the case study dataset we are working on will serve as input of a decision tree algorithm. Decision trees are supervised algorithms that use a tree-like model of decisions. The goal is to create a model that predicts the value of a feature by analyzing several input features.

As can be seen in Table 5 (shown at the end of the paper), feature #11 (`actionComponents`) is the objective field. In this case, we would like to know which components are suggested to new users so they can add them in the ENIA mashup application workspace. This microservice does not need any feedback from the decision tree algorithm.

Table 6 Datasets created for each user profile

Inst.	#1 idUser	#2 userType	#3 isNewUser	#4 operationPerf	#5 deviceType	#6 interForm	#7 partOfTheDay	#8 Season	#9 Country	#10 Weather	#11 objective field actionComponents
Features											
Dataset 1: Farmers											
#1	432	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Clouds	CuttleRoads
#2	432	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Clouds	BiosphereReserves
#3	120	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Rain	Wetlands
#4	120	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Rain	Weather
Dataset 2: Politicians											
#1	511	Politician	Yes	Add	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#2	511	Politician	Yes	AddGroup	Smartphone	Touch	Afternoon	Spring	Spain	Clear	GeoParks
Dataset 3: Tourists											
#1	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#2	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Weather
#3	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Facebook

Table 7 Datasets created for each user profile

Inst.	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11 objective field
	idUser	userType	isNewUser	operationPerf	deviceType	interForm	partOfTheDay	Season	Country	Weather	actionComponents
Features											
Dataset 1: User 17											
#1	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#2	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Weather
#3	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Facebook
Dataset 2: User 120											
#1	120	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Rain	Wetlands
#2	120	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Rain	Weather
Dataset 3: User 432											
#1	432	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Clouds	CuttleRoads
#2	432	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Clouds	BiosphereReserves
Dataset 3: Tourists											
#1	511	Politician	Yes	Add	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#2	511	Politician	Yes	AddGroup	Smartphone	Touch	Afternoon	Spring	Spain	Clear	GeoParks

There could be more microservices interested in receiving feedback from algorithms. Specifically, there is another microservice that creates a dataset with data of users' interactions over the mashup and seeks to categorize the users in accordance with some similarities. Clearly, the datasets will serve as an input for a clustering analysis algorithm. Cluster analysis is unsupervised algorithm that, given a set of users, tries to group them into clusters. All users in a cluster are similar to each other. For this microservice it is interesting to receive the algorithm feedback. This way, it can be offered in a web service how users are categorized according to the clustering analysis to other microservices or third-party clients. The microservice 'suggesting components to new users' could be interested in the feedback from the clustering analysis algorithms that is stored in the microservice that prepares the inputs for that algorithm.

As previously advanced, a dataset can be divided into several datasets conforming to the user profile. Now, users are also categorized on the basis of the clustering algorithms and it is possible to know in which cluster each user is contained by making a request to the microservice REST API. In Table 8 (at the end of the paper) we can see that a new feature #0 (`cluster`) has been added and two datasets have been created given such categorization.

As can be seen, the initial dataset acquired from the database has been strongly transformed by the feature engineering process. The new datasets are optimized to obtain the best possible results when applying machine learning algorithms and to create the best possible prediction models. Table 9 synthesizes the transformation that has been taken place in each feature.

- Feature `idUser` is kept (except in those datasets in which the components added by specific users had been analyzed).
- Feature `userType` is also kept (again, except in those datasets in which the components had been analyzed).
- Features `idSession` and `idInteraction` have been transformed in a new boolean feature `isNewUser`.
- Feature `operationPerformed` has been kept, but the instances with values of that feature different than `add`, `addGroup` have been deleted.
- Feature `dateTime` has been split into the features `partofTheDay` and `season`, both of which are also discrete features.
- `Latitude` and `Longitude` features have been merged into a new discrete feature called `country`.
- Feature `idComponents` which is an array has been transformed into a simple feature `idComponent`, and, as a consequence, the instances that had more than one value of this feature has been split into several instances.
- Finally, a new feature called `cluster` has been created and added to the dataset after a request to another

Table 8 Datasets created for each cluster identified in clustering analysis

Inst.	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11 objective field
	Cluster	idUser	userType	isNewUser	operationPerf	deviceType	interForm	partOfTheDay	Season	Country	Weather	actionComponents
Features												
Dataset 1: Cluster 1												
#1	Cluster 1	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	BeachTemperatures
#2	Cluster 1	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Weather
#3	Cluster 1	17	Tourist	Yes	Add	Tablet	Voice	Afternoon	Spring	Spain	Sunny	Facebook
#4	Cluster 1	120	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Rain	Wetlands
#5	Cluster 1	120	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Rain	Weather
#6	Cluster 1	432	Farmer	No	Add	Laptop	Mouse	Morning	Spring	United States	Clouds	CuttleRoads
#7	Cluster 1	432	Farmer	No	AddGroup	Laptop	Mouse	Morning	Spring	United States	Clouds	BiosphereReserves
Dataset 2: Cluster 2												
#1	Cluster 2	511	Politician	Yes	Add	Smartphone	Touch	Afternoon	Spring	Spain	Clear	WordHeritage
#2	Cluster 2	511	Politician	Yes	AddGroup	Smartphone	Touch	Afternoon	Spring	Spain	Clear	GeoParks

microservice who was informed about users categorized by the feedback from a clustering analysis algorithm.

5 Related works

The popularity of cross-device applications has led to an increase in works related to manage and to improve the accessibility and usability of distributed user interfaces in different contexts. Existing works focus on different kinds of applications. For instance, Han et al. [17] studied web searches where people initialized them in one device but completed them in another (PC to mobile or vice versa). This article is mainly focused on mobile touch interactions, whereas we intend to cover any type of interaction in different kinds of devices.

Kane et al. [22] also focused on the mobile platform and studied web browsing activities between PCs and mobile devices to enhance the user experience in mobile devices by facilitating fast access to URLs visited in PCs. However, the information obtained from the users' activities is not stored in a structured database and it is not processed by using feature engineering mechanisms.

Albertos-Marco et al. [1] describe how to distribute the interaction over cross-device applications, especially in Internet of things (IoT) related applications, by using responsive web design user interfaces. Regarding this study, our approach can be related to the producer-consumer and parallel patterns within the responsive cross-device applications (RCDA) domain, since the interactions are produced in the UIs and consumed by our microservice-based architecture, and many devices are coordinated for simultaneous use [32].

The interactions over mashups and users' behavior can be analyzed to improve the interface customization, task automation and the selection of the most suitable components, among other applications [16]. But there is still a need for a complete and personalized user interaction, where specific solutions can be provided to the users needs [28]. In [36], the authors present some facilities to find out the potential and actual behavior loops in publish-subscribe-based mashup UIs, solving the analysis of the direct and indirect interaction. Direct interaction refers to those events activated by the component itself; indirect interaction is initiated by other components. In our approach, we manage both types of interactions since the storage of the information related to the interaction can be done both by the component itself and by external elements, whenever the call is made to the corresponding microservice.

Research opportunities have arisen in order to manage the huge increase in cross-device applications over the last years by creating architectures that support these applications. Sanctorum et al. [31] propose an approach that empowers end users to create, modify and configure their own

Table 9 Features transformation in the feature engineering process

Before FE	After FE	Consideration
IdUser	IdUser	Feature kept or deleted
UserType	UserType	Feature kept or deleted
IdSession	IsNewUser	New boolean feature created
IdInteraction		
OperationPerformed	OperationPerformed	Feature kept (filter = add or addGroup)
DeviceType	DeviceType	Feature kept
InteractionForm	InteractionForm	Feature kept
DateTime	Season	New discrete feature created
	PartOfTheDay	New discrete feature created
Latitude	Country	New discrete feature created
Longitude		
Clustering feedback	Cluster	New discrete feature created
IdComponentArray	IdComponent	Objective field

distributed user interfaces by an architecture that provides the functionality for the synchronization of UIs on different devices using a distributed model-view-controller (MVC); however, the approach is limited to the analysis and design stages, and it lacks a real implementation. Fehling et al. [10] cover different possibilities of architectural patterns in cloud environments. These architectural patterns allow us to design and build multi-devices applications shared by multiple customers that can be offered as configurable cloud services on elastic infrastructures. In our case, we combine the advantages of cloud infrastructures and service-oriented architectures to propose an architecture that manages user interaction with distributed cross-device user applications.

Finally, mashups are widespread nowadays in several scopes. Some commercial examples are Geckoboard [15], a KPI (key performance indicator, i.e., a business metric for evaluating factors that are crucial to the success of an organization) dashboard where users can visualize and work with their most important business data in real time; Cyfe [6], that allows users to build their own dashboard adding pieces of information through social media, analytics, sales or project management components among others; Freeboard [14], a customizable dashboard focused on the IoT devices; and Netvibes [26], that analyzes data through customized components. Although they can be deployed on different devices and platforms, the components of these mashups are isolated from each other. In contrast, our application domain ENIA provides mashup UIs with interrelated components, which implies going further in the analysis of the interaction.

6 Conclusions and future work

This paper has presented a microservice-based architecture that exploits the interaction data generated by users when interacting with cross-device mashup applications through multiple forms of interaction. Each microservice of this architecture has a concrete purpose and queries a relational database that contains data about the user behavior. Usually, the interaction data fetched are too raw and, in order to create optimal datasets for further analysis with machine learning algorithms, a deep feature engineering work is performed.

The new optimized datasets increase the possibilities of creating accurate prediction models that might help to improve the user experience when using the mashup application through heterogeneous devices. This can be done by stepping to the users' needs and providing them with a customized user experience. It is not only considered the user device and form of interaction in a concrete moment, but the customization carried out according to the study of the user behavior and some other user's behavior in the mashup application.

Microservices are equipped with REST API web services for exposing their functionalities to other microservices, receiving feedback from algorithms or other sources, and communicate with third-party clients. The process of creating datasets by the proposed architecture is autonomous and continuous; hence, datasets are always updated with the latest interaction data.

An interesting forthcoming in the microservices architecture would be the development of an API gateway that facilitates the access of third-party clients to the functionality offered by the microservices. Also, it will help other microservices to explore and discover functionalities that already exist in the architecture.

The set of benefits of the proposed approach include (they are not limited to, though):

- Multiple devices accessing simultaneously to the application by using concurrent sessions are supported.
- Multiples forms of interaction are also supported. This is important due to the fact that each device can present a different form of interaction and even some devices can have several of them.
- The architecture that supports the cross-device interaction run in the cloud so that many users can simultaneously access the application from everywhere at anytime and the scalability is guaranteed.
- It is ready to work with new devices that incorporate natural user interfaces (NUIs) since the device and form of interaction are taken into consideration.

- Different user profiles and categories can be handled as well as there can be guest users. This classification significantly influences on predictive models.
- It can handle the context awareness capability of some devices in ubiquitous computing system which is taken care of to enhance the user experience.
- The microservice granularity allows us to easily design, implement and deploy microservices that obtain datasets with different purposes that easily escalate according to their needs.
- The microservice-based architecture allows developer teams to simultaneously implement new functionalities independently. Also, each team can use the technologies that best adapt to solve the problem and should easily be integrated in the system.
- Even when the proposed architecture may be general, it is specially designed for component-based user interfaces that are trendy today due to its easiness of being configured by users and filled with cloud stored-services.
- Communication between microservices is solved by including a set of API REST web services.
- To prevent third-party applications or entities from accessing the internal microservices structure of the architecture, an API gateway can be implemented to canalize communications with other parties.
- Datasets are always updated thanks to an automated system that periodically generates new datasets. Also, datasets can be generated on demand by calling a web service in each microservice.
- Due to the fact that datasets are always updated, if the machine learning prediction models are periodically retrained the system can evolve over time, providing adaptation to new users, components, devices and forms of interaction.

A future development includes the use of datasets generated by the microservice architecture as input for machine learning experiments. From the exploitation of these data, some new insights can arise, such insights might be inferred into knowledge which can be applied over the work shown at Criado et al. [4, 5], where component-based interfaces are adapted at run time by using model transformation according to a set of rules. The new knowledge inferred by the experiments can update the rules repository, allowing the mashup application to autonomously evolve over time [11] and leading to a better user experience. As future work we plan to perform a study to measure and analyze the improvement of the user experience and usability.

Acknowledgements This work has been funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Project TIN2013-41576-R. A.J. Fernandez-Garcia has been funded by a FPI Grant BES-2014-067974. J.Z. Wang was funded by the US National Science Foundation under Grant No. 1027854.

References

1. Albertos-Marco, F., Penichet, V.M.R., Gallud, J.A.: Distributing interaction in responsive cross-device applications. In: Current Trends in Web Engineering ICWE 2016. LNCS 9881, pp. 174–178. Springer (2016)
2. Beck, K., et al.: Manifesto for agile software development. http://agilemanifesto.org
3. Chen, L.: Continuous delivery: huge benefits, but challenges too. IEEE Softw. **32**(2), 50–54 (2015)
4. Criado, J., Rodríguez-Gracia, D., Iribarne, L., Padilla, N.: Toward the adaptation of component-based architectures by model transformation: behind smart user interfaces. Softw. Pract. Exp. **45**(12), 1677–1718 (2015)
5. Criado, J., Vicente-Chicote, C., Padilla, N., Iribarne, L.: A model-driven approach to graphical user interface runtime adaptation. In: 5th International Workshop on Models, CEUR-WS, vol. 641, pp. 49–59 (2010)
6. Cyfe: Business mashup to manage social media, analytics, marketing, sales, support and infrastructure components. http://www.cyfe.com/
7. Daniel, F., Matera, M.: Mashups: Concepts, Models and Architectures. Springer, Berlin (2014)
8. Elmqvist, N.: Distributed User Interfaces: State of the Art, Distributed User Interfaces, Human–Computer Interaction, pp. 1–12. Springer, London (2011)
9. ENIA: The Environmental Information Agent Project. http://acg.ual.es/projects/enia/ui/
10. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud application architecture patterns. In: Cloud computing patterns: fundamentals to design, build, and manage cloud applications, pp. 151–238. Springer, Vienna (2014). https://doi.org/10.1007/978-3-7091-1568-8_4
11 Fernández-García, A.J., Iribarne, L., Corral, A., Wang, J.Z.: Evolving mashup interfaces using a distributed machine learning and model transformation methodology. In: OTM 2015. LNCS 9416, pp. 401–410. Springer (2015)
12 Fernández-Villamor, J.I., Iglesias, C.A., Garijo, M.: Microservices—lightweight service descriptions for REST architectural style. ICAART **2010**, 576–579 (2010)
13 Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
14 Freeboard: Dashboards for the internet of things. http://freeboard.io/
15 Geckoboard: KPI mashup dashboard software for businesses. https://www.geckoboard.com/
16 Ghiani, G., Paternò, F., Spano, L.D., Pintori, G.: An environment for end-user development of web mashups. Int. J. Hum. Comput. Stud. **87**, 38–64 (2016)
17 Han, S., He, D., Yue, Z., Brusilovsky, P.: Supporting cross-device web search with social navigation-based mobile touch interactions. In: User Modeling, Adaptation and Personalization UMAP 2015. LNCS 9146, pp. 143–155. Springer (2015)
18 Hoyer, V., Fischer, M.: Market overview of enterprise mashup tools. In: Service-Oriented Computing ICSOC 2008. LNCS, vol. 5364, pp. 708–721. Springer (2008)
19 Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley, Reading (2011)
20 Inc: Cisco Systems. Cisco Visual Networking Index: Forecast and Methodology, 2015–2020. Technical Report, Cisco Systems, Inc. (2016)
21 Iribarne, L., Padilla, N., Criado, J., Vicente-Chicote, C.: Metamodeling the structure and interaction behavior of cooperative

component-based user interfaces. J. Univ. Comput. **18**(19), 2669–2685 (2012)
22 Kane, S.K., Karlson, A.K., Meyers, B.R., Johns, P., Jacobs, A., Smith, G.: Exploring cross-device web use on PCs and mobile devices. In: Human–Computer Interaction INTERACT 2009. LNCS, vol. 5726, pp. 722–735. Springer, Berlin (2009)
23 Kantardzic, M.: Data Mining: Concepts, Models, Methods and Algorithms. Wiley, London (2002)
24 Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html (2014)
25 Nebeling, M., Zimmerli, C., Husmann, M., Simmen, D.E., Norrie, M.C.: Information concepts for cross-device applications. In: 3rd Work. on Distributed User Interfaces: Models, Methods and Tools (DUI 2013), pp. 14–17 (2013)
26 Netvibes: Mashup application for analyze and act on all the data that matters to a brand or business. https://www.netvibes.com/
27 OGC: The open geospatial consortium. http://www.opengeospatial.org/
28 Paredes-Valverde, M.A., Alor-Hernández, G., Rodríguez-González, A., Valencia-García, R., Jiménez-Domingo, E.: A systematic review of tools, languages, and methodologies for mashup development. Softw. Pract. Exp. **45**(3), 365–397 (2015)
29 Poslad, S.: Ubiquitous Computing: Basics and Vision, pp. 1–40. Wiley, London (2009)
30 Ramasubramanian, K., Singh, A.: Feature Engineering, pp. 181–217. Apress, Berkeley (2017)
31 Sanctorum, A., Signer, B.: Towards user-defined cross-device interaction. In: Current Trends in Web Eng. ICWE 2016. LNCS, vol. 9881, pp. 179–187. Springer (2016)
32 Santosa, S., Wigdor, D.: A field study of multi-device workflows in distributed workspaces. In: The ACM Int. Joint Conf. on Pervasive and Ubiquitous Computing (UbiComp 2013), pp. 63–72. ACM, New York (2013)
33 SOAP (Simple Object Access Protocol) W3C Standard. https://www.w3.org/TR/soap12/
34 REDIAM network. The Andalusian Environmental Information Network, Spain. http://www.juntadeandalucia.es/medioambiente/site/rediam/
35 Thönes, J.: Microservices. IEEE Softw. **32**(1), 113–116 (2015)
36 Tschudnowsky A., Gaedke M.: Loop discovery in publish-subscribe-based user interface mashups. In: Engineering the Web in the Big Data Era ICWE 2015. LNCS, vol. 9114, pp. 683–686. Springer (2015)
37 Vallecillos, J., Criado, J., Fernández-García, A.J., Padilla, N., Iribarne, L.: A web services infrastructure for the management of mashup interfaces. In: Workshop on Engineering Service-Oriented Applications (WESOA 2015). LNCS, vol. 9586, pp. 64–75, Springer (2016)
38 Velosa, A., Schulte, W.R, Lheureux, B.J.: Hype cycle for the internet of things. https://www.gartner.com/doc/3371743/hype-cycle-internet-things- (2016)
39 zur Muehlen, M., Nickerson, J., Swenson, K.D.: Developing web services choreography standards: the case of REST vs. SOAP. Decis. Supp. Syst. **40**(1), 9–29 (2005)