The Journal of Systems and Software 213 (2024) 112033

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A study on creating energy efficient cloud-connected user applications using the RMVRVM paradigm[☆]

Lavneet Singh[a,*], Saurabh Tiwari[a], Sanjay Srivastava[b]

[a] Software Engineering Research Lab, DA-IICT, Gandhinagar, India
[b] DA-IICT, Gandhinagar, India

ARTICLE INFO

Keywords:
Green software
Energy efficient client applications
Battery saving
Non-functional requirements

ABSTRACT

Many applications that run on smartphones are heavy on User Interface (UI) and depend on back-end services deployed on the cloud to fetch the required data through REST-based API. Because of the large number of devices actively being used, their collective energy consumption is very significant. Saving energy on these devices is beneficial not only for reducing the carbon footprint but also for the end users as it results in longer battery life. The data fetched by these applications using the REST API is generally processed, filtered and made compliant with the user interface through a paradigm called the Model View View-Model (MVVM).

In our previous work Singh (2022a), we proposed a novel Remote-Model View Remote-View-Model (RMVRVM) paradigm to reduce battery consumption by MVVM-based UI-centric cloud-connected applications. In this paper, we present new results and details of the architecture and properties of the RMVRVM paradigm, discuss the complexity shift, provide a migration framework to help practitioners migrate the existing MVVM-based applications and also discuss the possibility of RMVRVM replacing MVVM. We apply the migration framework on an open-source MVVM application and run the experiment to prove the efficacy of RMVRVM. We also expanded the experiments to include the iOS platform and 4G network. To cover the complexity shift to server-side, we discuss the energy consumption on the cloud and how energy-saving practices in cloud data centers help save overall energy consumption.

1. Introduction

The applications that run on various devices, like smartphones and smartwatches, are increasingly connected to the cloud to obtain data that these applications need for their operations. These applications have, in general, substantial user interface (UI) complexity and the data obtained from the cloud comes in via requests to the REST API-based services running on the cloud as the back-end of these applications. The devices running these applications have limited resources like the battery. The battery gets consumed (Carroll et al., 2010) when these applications perform activities like obtaining data from the cloud, processing the data received to get the required information, displaying it to the user etc. The applications use the REST API to communicate with their back-end systems (IBM, 2022) running on the cloud. In most cases, these APIs have already been defined and developed, for example, for use by the desktop or the web applications developed. When used by applications in battery-constrained devices like smartphones, the same API sends more than the required data that is then processed by the smartphones. This consumes their battery unnecessarily. This has two effects — one, the battery gets drained sooner, and two, the application's response time increases. Both these effects result in poor user experience because users expect their smartphone's batteries to last as long as possible and the applications to perform frugally.

The developers of UI-heavy applications use paradigms like Model View View-Model (MVVM) (Smith, 2009; Kouraklis, 2016) to bring data into the UI elements that the users see on the screens from the application's underlying data models. The paradigms like MVVM define the view-models that are replicas of the state of UI elements. They bind the view-model objects to their UI elements using data binding (Android Jetpack, 2022). Any update to the state of the view-model object automatically reflects in the UI elements bound to the view-model. Inversely, any change in the state of the UI elements, most likely done by the user, is reflected in the view-model of the UI elements. Generally, one view-model is defined for each page of the application's UI. Because the view-models are tightly bound to the UI elements, the data models, which receive data from the application's back-end

☆ Editor: Prof Raffaela Mirandola.
* Corresponding author.
E-mail addresses: lavneet_singh@daiict.ac.in (L. Singh), saurabh_t@daiict.ac.in (S. Tiwari), sanjay_srivastava@daiict.ac.in (S. Srivastava).
URL: https://sites.google.com/view/saurabhtiwari (S. Tiwari).

https://doi.org/10.1016/j.jss.2024.112033
Received 1 August 2022; Received in revised form 8 December 2023; Accepted 19 March 2024
Available online 25 March 2024

services running on the cloud, need to be processed on the device by the application to extract the state that matches the view-model and then copy the extracted state to the view-model. Thus, using the MVVM paradigm on battery-constrained devices allows the applications to use the CPU to perform data transformations and drain the battery. Further, the REST API that the application uses sends data to it, which may be only a fraction, which is valuable for the application. Processing received data to extract the helpful fraction and populate the view-models using the extracted data leads to higher battery consumption on smartphones (Pramanik et al., 2019). More data on the device leads to higher memory space usage, further consuming the power.

The previous work of the authors (Singh, 2022a) proposed the RMVRVM (Remote-Model View Remote-View-Model) paradigm to replace the MVVM paradigm in most cloud-connected UI-heavy applications on the client-side. The new paradigm shifts the complexity of the state, represented by the model and its conversion to and from view-models, onto the back-end of the application that runs on the cloud. Because of the move of model and view-models to the back-end, the requirement for client-side data processing was reduced substantially. The REST API is modified to send only the required data to the client-side and nothing more. The result is that the application consumes less battery and becomes agile and frugal because it has much less data to process on the client-side. Both these results are desirable by the user and help write green software by reducing the carbon footprint of the application.

This paper extends the work of Singh (2022a) in two ways. First, by providing detailed information about the paradigm both from the server and client-side. Second, by providing a migration framework for practitioners to follow for migrating their MVVM-based applications to RMVRVM. We apply this migration framework on an open-source MVVM application and migrate it successfully to RMVRVM. The gains in energy saving and better response time on the migrated application prove the efficacy of the RMVRVM paradigm. We have also conducted experiments on the iOS platform and 4G network. Furthermore, in RMVRVM, the complexity of the application is shifted to the cloud. Hence, in this paper, we discuss the energy consumption in cloud data centres and how the practices to conserve energy could help reduce total energy consumption by RMVRVM-based applications. Specifically, the contributions of this paper are as follows:

- The RMVRVM paradigm's architecture and properties.
- A framework to help practitioners migrate MVVM applications to RMVRVM without significant manual effort.
- Successful application of the migration framework to an open-source application and study its before and after energy consumption and response time.
- Experiments on Android and iOS platforms with 4G and WiFi connections.
- Discuss energy consumption by the back-end services in the RMVRVM paradigm on the cloud data centres.

The remainder of this paper is organized as follows. Section 2 provides background information on the current architecture of such cloud-connected applications on both the client and server-side. Section 3 covers the related work. Section 4 introduces the paradigm, discusses its architecture and properties, discusses server-side energy consumption and provides the migration framework. Section 5 describes the experiments and case studies conducted in the industry to implement the paradigm. Section 6 provides the conclusion and plan of work in the future.

2. MVVM — The current state-of-the-art

This section discusses the current state-of-the-art architecture of UI-heavy applications connected to the cloud to obtain data using the REST API. The discussion focuses on the client-side MVVM paradigm's details.

Fig. 1 shows architecture (Smith, 2022) for UI-heavy application development targeted to devices like smartphones. The illustration uses the example of the Microsoft Azure cloud service. There can be different kinds of applications on the client-side — smartphone applications, applications running on smart devices like smartwatches, IoT devices connected to their base stations, inside robots, desktops, and progressive web apps (PWA). All of these applications could be consuming the same REST API. The data useful for these applications reside in the cloud, perhaps sent over to the cloud by other sources or applications. This data is obtained by these applications using the REST API calls. These REST APIs are defined by the back-end of these applications running on the cloud. Multiple applications could share common REST APIs. The server-side application's architecture on the cloud generally consists of different cloud services to receive and process requests. The web server on a cloud data center uses services like Azure App Service. The API Gateway layer receives the request from the client-side. It could be directly received from the web server as well. The API Controllers are responsible for processing the request further. It could be authenticated through services like Azure Active Directory and Azure KeyVault for obtaining secrets like passwords. The data contained in the request could be processed further using cloud services like message queues and background task execution like function apps. The data is further processed to get the information requested in the client-side request. These steps generally involve querying the database to obtain and process information through the database layer. It could be using either SQL or NoSQL database services provided by the cloud service provider. It could also have integration with any 3rd-party APIs to get additional data. The combined data could then be returned to the client as a response to the incoming request through the API Gateway layer.

The client-side of the application receives data sent as the response to the API request it sent earlier. The response received by the application would require the transformation of the data model objects. It may require filtering, sorting or other kinds of processing before the final desired state of the data model is achieved. Fig. 2 represents the MVVM paradigm design responsible for doing these steps on the client-side. It represents an implementation of a typical UI-heavy smartphone application where the MVVM paradigm is realized through various layers of the front-end code using objects like models, view-models and transformations between them as the data is received from or sent to the back-end through the API calls. Typically, a UI-heavy application has many UI pages. The user could navigate through these pages. Each UI page has its state represented by its view-model, i.e., a one-to-one mapping exists between the view-model and its UI page. Further, the view-model object has properties that map one-to-one to each element of the UI page. This mapping is achieved by data binding, where the MVVM allows the UI control to obtain its state from the underlying view-model property. Data binding allows automatic updates to the view-model as soon as the user provides input to the UI elements of the view bound to the view-model. This is one-way data binding.

Furthermore, there is two-way data binding available too. In this, the UI page is refreshed automatically when there is a change in the properties of its view-model object. The one-way and two-way data bindings can co-exist in the same application because the data binding is between a UI element and its view-model element. In Fig. 2, the first two UI page blocks have two-way data binding with their view-models, depicted by the double-sided arrows. The third page, however, has one-way data binding depicted by a one-sided arrow. Thus, the MVVM paradigm enables automatic updates of the UI state based on changes to the underlying data in the view models and vice-versa. Additionally, the states of UI elements could be bound to each other; for example, the rotation of a text field is bound to the slider such that a user can rotate the box by sliding the notch on the slider control. The MVVM paradigm is a valuable software design for creating UI-heavy client applications.

The converters tie the view-models and models in the MVVM paradigm. These converter objects play an essential role in the implementation of MVVM. Their job is to convert the data to and from

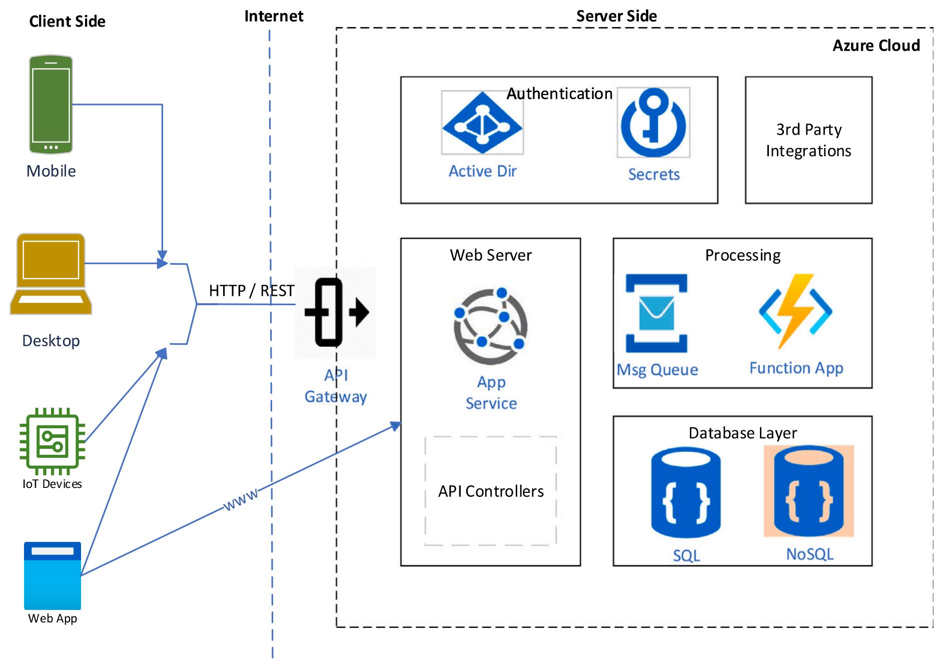


Fig. 1. REST API based cloud application architecture.

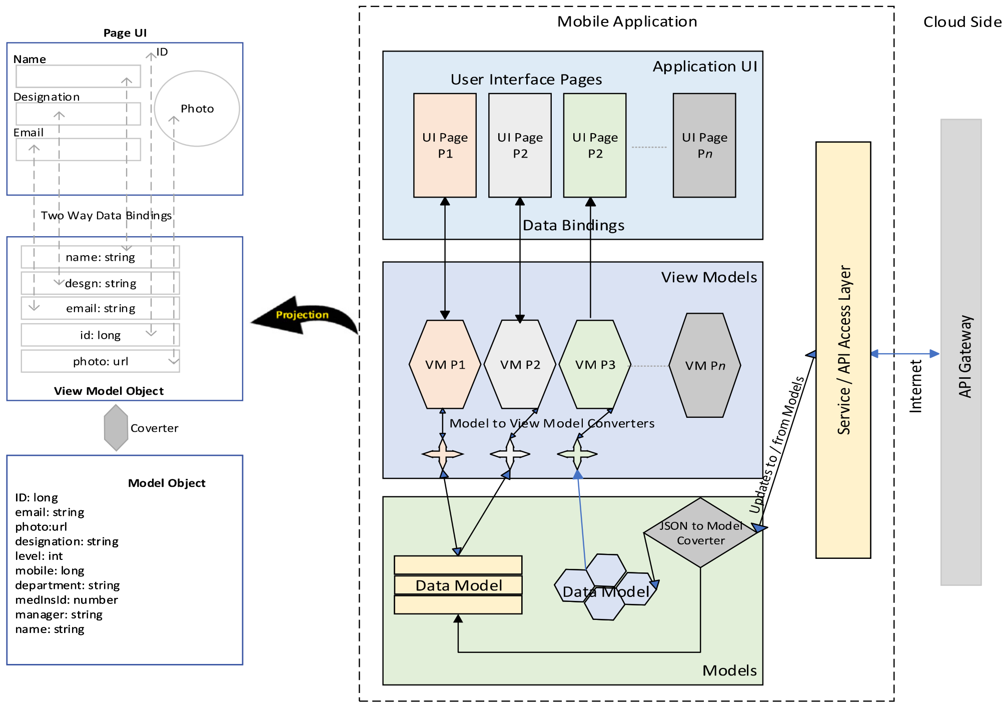


Fig. 2. Client-side application design.

models to view-models. These converters could be as simple as a subset of properties of model objects or a separate class that appropriately processes model object data to create a view-model object. There is either one-to-one or one-to-many mapping between the model and view-models. A model object is often helpful to multiple view-models because the UI could show different views of the same data. Therefore, view-models of each UI page could be using a single model as their data source. If the state of the data model changes because new data has arrived from a response to the API call, the view-models get notified of the state change with patterns like publish-subscribe. Once the view-models get the notification, they extract the data from models as per their structure, and this change in their state is reflected in the UI through the data binding mechanism. Conversely, when the user inputs data in the UI elements and confirms/saves it, the view-models get updated due to data binding. The view-models have direct reference to the data models, so they can immediately invoke the data model to update its state. The data race condition, where multiple view-models race to update the same model, does not arise because only one UI page is visible to the user at a time.

The format of the response received by the client from API invocation needs to be converted to the models. JSON is a popular format used for state transfer in REST API applications. Most UI frameworks support converting the response received in JSON into model object states. If the response has excess data or is a mismatch to the model objects, the service on the client-side that receives the response needs to process the data to convert it into a structure matching the model objects and then update/create the model object states. The service layer is also responsible for sending the request to the REST API by converting the model objects to the HTTP request body and configuring the rest of the request structures like the headers, query parameters and the base URL.

The projection in Fig. 2 illustrates the MVVM paradigm construction with a sample. The top box depicts a UI page with few details about the employee, specifically, ID, Name, Designation, Email and photo. This set of fields is only a subset of a large data model object that represents the employee with many more properties. The view-model object has one property corresponding to one UI element on the page. These properties are data bound to the respective UI elements. The view-model data binding to UI elements allows the automatic update to the UI elements if there is a change in the bound property and vice-versa. Most contemporary UI frameworks support this data binding feature. A converter is required to convert the complex object of the data model to a view-model object. Note that the converter could map the corresponding properties between the objects or have more data processing to make it appropriate for the view-model. For example, if the name in the UI consists of first and last fields, it could require that the name property of the data object is parsed, and first and last names are extracted as separate values and then put in the properties of the view-model object.

2.1. MVVM architecture — issues on resource-constrained devices

The MVVM paradigm, used for developing cloud-connected applications that run on resource-constrained devices like smartphones and smartwatches, is not resource-friendly. The MVVM architecture was designed without factoring in the power consumption on resource-constrained devices. It is well suited for applications that run on conventional devices, like desktops and web browsers, connected to the cloud, where power consumption is not constrained. This subsection discusses the issues of using the MVVM paradigm in resource-constrained devices.

2.1.1. Frequent conversion of data between models and view-models

In the MVVM paradigm, there is frequent, automatic, and back-and-forth data conversion from model to view-model using view-model converters. This conversion consumes the CPU and, hence, the battery. The model objects could be a data source for many view-model objects due to their one-to-many mapping to view-models. The structure and size of the model objects generally are complicated and extensive. The converters work on these large objects to extract data from and put in view-models and vice versa. The extent to which the battery is consumed depends upon the frequency of conversion of data and the complexity of the conversion code. The frequency depends on the application usage and the use cases it covers. An application UI that navigates through different pages and presents a different view of data on a mobile phone will tend to consume more battery.

2.1.2. Responses of REST API calls not tailored to the view-models

The response object received by the client-side should be tailor-made to the view-models. However, that is not generally the case in MVVM. This could happen for a variety of reasons: the APIs being designed by a separate team using generic requirements that apply to both desktop and smartphone applications, the API serves more than one kind of client application, or there being a communication gap between the teams that handle back-end and the front-end of the application. Many smartphone applications are launched much later than websites or desktop applications. In such cases, the APIs used by the smartphone application are likely the same ones developed earlier for other applications. The result is that the responses that the smartphone application receives from the REST API are not suitable for them. This necessitates further processing of the response by the smartphone application to make the data suitable for the model objects.

2.1.3. Unutilized model objects and collections

Generally, an application's UI on a smartphone is much simpler because of lesser screen real-estate than its desktop or web versions. Hence, the view-model objects use only a subset of the data and collections stored in the model objects. Because of this structure mismatch, the converter objects are needed. They transform the data from models to view-models and vice-versa when the application is running and receiving data from users and responses from the REST API calls. Additionally, as the application runs and the user interacts with it, there is constant shuffling, transformation and filtering done on the client-side. All these activities by the application result in CPU and memory utilization, leading to battery consumption. This constant data processing negatively impacts the application's responsiveness too, leading to poor user experience.

3. Related work

The work in Carroll et al. (2010) analyzes how different hardware components in a smartphone contribute to battery consumption. It finds that the CPU is one of the most energy-consuming hardware in the phone. CPU power consumption closely follows the power consumption by the GSM module. A report on battery consumption in Android devices by Pramanik et al. (2019) reports that the applications running on phones consume up to 24% of battery. Alshurafa et al. (2014) applied an optimization technique to the application that monitors health activity using smartphone sensors. The technique uses intervals of inactivity and makes the application/sensor go into sleep mode to reduce battery consumption. The technique attempts to address the problem within the given architecture by moving the application to sleep mode when possible - this is similar to reducing CPU usage by programming tricks while staying within the MVVM paradigm. The work by Kim et al. (2012) identifies unnecessary data reception as a problem. The author proposes sending the battery level values to the server and trying to reduce the payload size sent to the client-side. The 4G network usage and how the power consumption is linked to it in the application is studied by Gupta et al. (2011). The author studies power consumption patterns in popular applications that use 4G networks. Another design where the background applications are stopped, called Application State Proxy (ASP), is proposed by Bolla et al. (2014).

Chen et al. (2012) proposes offloading tasks from an Android application to an IaaS-based solution on the cloud, like a virtual machine. However, the work does not consider the overall mobile architecture of the application. In work by Xia et al. (2014), a Hadoop map-reduce engine-based offloading mechanism is proposed for mobile applications. The primary goal is to reduce the number of computation tasks run on the phone. The decision whether to offload the task to the cloud or not is made on the device. This is still problematic because the decision-making will consume energy. The work by Akherfi et al. (2018) assumes that the tasks to be offloaded to the cloud are already known and, therefore, devises a technique to partition the application at the compile time. The partition is responsible for task execution and is run on the cloud. Ragona et al. (2015) consider wearables as smart devices and propose a technique to offload tasks to the paired smartphone for computation. This does not consider the GSM enable wearables that can operate independently of the phone's presence nearby.

4. The RMVRVM paradigm

This section discusses the RMVRVM paradigm with details about its architecture on the cloud and its working, as well as the simple client-side design due to reduced complexity. The complexity shift from client to server-side is explained in the following subsection. Because the complexity is moved to the cloud, we discuss the server-side energy consumption on a cloud data center.

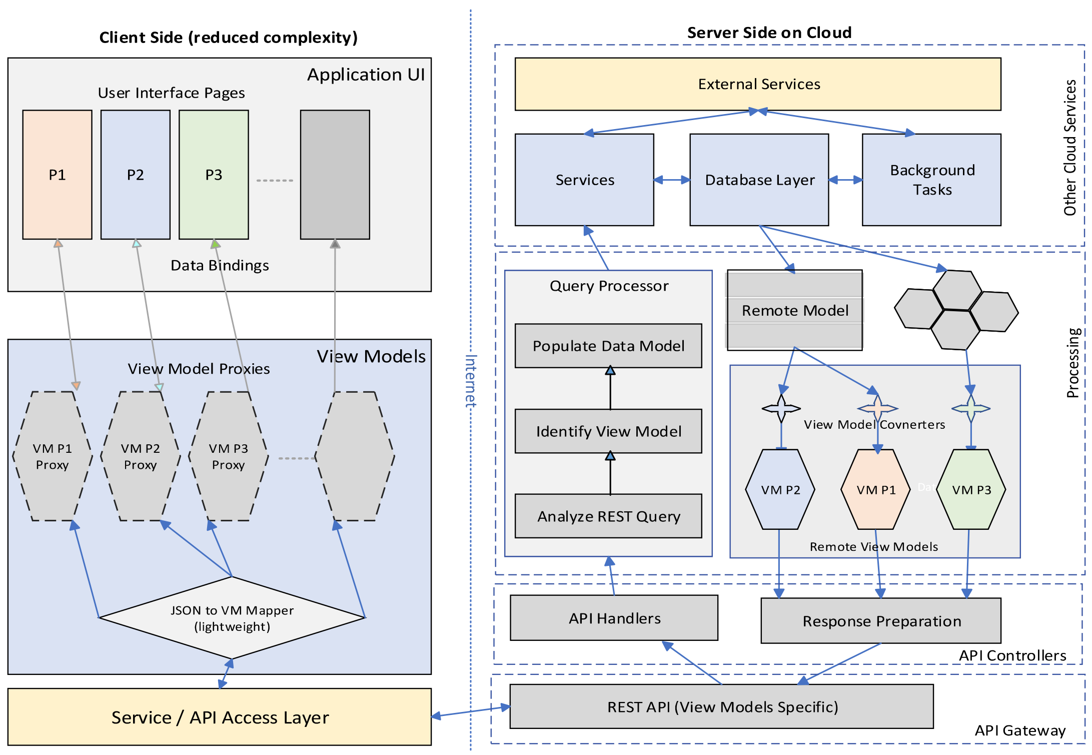


Fig. 3. RMVRVM architecture.

4.1. *Architecture of RMVRVM paradigm*

The architecture of the RMVRVM paradigm significantly differs from the MVVM paradigm, even though it takes its name from the latter. The RMVRVM paradigm moves the UI-heavy application's model and view-model objects from the client to the server-side. This move profoundly impacts the application's architecture, and the resultant architecture varies significantly from the traditional MVVM architecture.

Fig. 3 shows the application architecture of the RMVRVM paradigm. The model and view-model objects, including the converters, are moved to the server-side. The REST API is changed in its signature and response to match the view-model proxies' structure on the client-side. In the RMVRVM paradigm, handling the complexity of the application is put on the cloud, and the client-side of the application that runs on resource-constrained devices like smartphones is simplified to reduce energy consumption due to the CPU, memory and network to the minimum. It is possible because of the elimination of (1) excess data coming to client-side thus reducing the energy consumed by the network, (2) processing of data by the CPU as the data already is a match to the view-model objects, (3) the role of converters because the model objects are absent on the client-side. This results in significant battery savings for smartphones. There are valuable gains for the users because their smartphone's battery now runs longer, and the application is more frugal and a delight to use. The details of the RMVRVM architecture are discussed in the following subsection.

4.1.1. *The design on server-side*

The server-side design involves processing the data that the REST API receives, creating appropriate view-model objects and then querying the model objects that further query the database to update their state first. Then, the state of view-model objects is updated. The view-model objects are then serialized into JSON and sent back to the client in the response object of the REST API.

API Gateway

The API Gateway exposes the API to the clients of the server-side application. It is a cloud service that aggregates the server-side API and hides the details of server-side components from the client. Using the API Gateway, the architecture encapsulates the implementation details from clients. In the RMVRVM paradigm, the API conforms closely to the requirements of the client-side view-model proxy objects. The API is designed so that the response JSON matches the structure and types of properties of the UI page's view-model for GET requests. The POST or PUT requests the clients use to send data to the server would match the form of view-model objects. That way, the client-side does not have to process the data to match the API's parameters, thus saving the battery. The API Gateway cloud service can be configured to define the view-model-specific API layer while keeping the old API structure intact. The API Gateway will transform the requests from the new view-model-specific API to the old API.

API Controller

The API Gateway receives the request. It could do a few more tasks on it, like decrypting the data. The API Gateway passes over the decrypted request further to the API Controllers. Typically, there is one controller for each API. That controller has the code to extract information from the request query parameters, the object from the request's body, their validation, and it sends information further for processing. If, during validation, any query parameter is found to be erroneous, the controller could return the request with an error code without passing it further down. The second responsibility of the API controller is response preparation.

Processing

The components under the processing group are primarily responsible for implementing the RMVRVM paradigm's core logic.

Query Processor

The API handler has validated the data reaching the Query Processor, so it does not need re-validation. The primary responsibility of the Query Processor is to figure out which view-model is to be instantiated based on the query parameters details. It analyzes the query for values and flags that identify the view-model. Then, the object of the view-model is created. The implementation could take advantage of the fact that the view-models of applications running on many devices are created in a central location in the cloud. Using the state of view-model objects cached in a cloud service cache system could reduce the time to make view-model objects available. In non-cached flow, when the view-model is unavailable in the cache, the query processor will create the model object of the view-model and ask it to update its state from the database. Then, the converter object will fill the view-model from the model's state. The cache system could be implemented for model objects as well.

Remote View-model

The query processor's view-model can be considered the remote view-model of the UI page on the client device. The view-model is tied to its underlying model through the publish-subscribe mechanism. As the view-model gets notified about the updates in the state of its model, it uses converters to get its state updated from its changed properties. The converter objects own the conversion details between the view-model and model objects. There is a one-to-one mapping between the converter and view-models. The model objects could serve as data sources for multiple view-models, so the relation between converters and models could be many-to-one. The converter object does not live independently of its view-model, their lifetimes are tied up.

Remote Model

The architecture's remote model object represents the model transferred from the client-side. However, because the model is at a central location in the cloud, the remote model does not need to match precisely the erstwhile model on the client-side. The developers are at liberty to keep the structure of the remote models the same as the original models or change them as the situation demands. Changing the model objects will affect the code in converter objects. The model objects can be created as singleton objects, keeping their state updated from the source of their data — the databases. Also, the model objects can be cached, and the frequency of queries to databases is reduced. Another advantage of moving the model to the cloud is that the model objects are no longer remote to their data source. The queries to the database are much faster than the API request response, which are otherwise used to transfer object model data from databases to the client-side where the model objects reside.

Model to View-model Converters

The converters transform the data from the model to its view model. A converter is an object owned by its view-model. The view-model references the converter object to the source of its data, the model object. The model invokes the converter and then converts the state of the model object into the state of the view-model object that owns it. The second mechanism that a view-model can use is to have reference to its model directly and invoke the converter object by itself once it receives the notification from the model object about the change in the model's state. In rare circumstances, it is also possible that a view-model could require data from more than one model. In such cases, the converter plays the integrator role, picking up data from multiple models to create the object state in the view-model. Once the converter transforms the data into view-model, it is recommended that there is no further processing done on the data by the view-model. This helps adhere the architecture to the single responsibility principle.

Response Preparation

The response preparation involves serializing data from a view model into the REST API response object. This response object contains the view-model object's state. The response object could copy the property values from the view-model into its structure or contain a reference to the view-model. The choice will depend on the design that the application developer chooses to implement. This component is responsible for preparing the REST API response data object and providing it to the API layer, which sends the response back after packaging it to the relevant response structure of the REST request. The response preparation generally involves creating JSON of the state of the view model, which needs to be serialized back to the client-side as a response to the REST API call. The structure of the JSON should only be governed by the view model's properties. The response preparation layer can change other parts and parameters of the response, such as setting the correct REST API response code and status code.

The RMVRVM paradigm advocates changing REST API so that each view-model has a separate API. The API response, that is, the JSON object embedded in the response object, should be a direct map to the properties of the view on the client-side, which the remote view-model represents on the server-side. This part of the design is crucial for functioning the new (or proposed) architecture. If the API surface refactoring is not possible due to backward compatibility or other constraints, several solutions can be employed, like using default parameters in the request headers. Only when the default parameter is set to a particular value does the RMVRVM mechanism trigger on the server-side; otherwise, it is bypassed, and the initial response can be returned.

4.1.2. *Working of the server-side*

The previous section provided details on the individual components of the server-side. In this section, we present end-to-end server-side execution details of an API request using a sequence diagram as shown in Fig. 4. The sequence diagram shows events in an example application that uses the RMVRVM paradigm. The application shown here is referred from Singh (2022a), an industrial case study I.

The client application calls the GetActivityData API. The API Handler receives this API request on the server-side deployed on the cloud. API Handler is equivalent to the API Controller in the architecture. API Handler processes the query and extracts the value of the unitID parameter passed in the REST API URL. Based on the value of this parameter, it creates an instance of the view model. On the client-side, each option has three views — day, week and month views of activity data in the health application.

Consequently, three remote view models can be created on the server-side. The sequence diagram depicts this using the swim lanes element of the unified modeling language (UML), showing that only one of the three view-model objects shall be created at a time. The view-model creates its converter and hooks up to the model. It then asks for the latest data from the model, which queries the database to refresh its state and then provides the latest data to the view-model converter. The view-model converter then converts the data to the exact form that fits the view-model object. Thus, the state of the view-model is updated, and response preparation can be initiated. The response preparation involves serialization of the view-model object's state into a JSON, putting this JSON into the REST API's response object and then sending this response object back to the client. On the client-side, the view-model proxy object's state is readily filled up without converting, filtering, or processing the JSON from the response. Hence, with the least effort on the client-side, the view is refreshed instantly because the view-model proxy is data-bound to the view.

4.1.3. *Client-side design*

The left half of Fig. 3 shows the client-side design in the RMVRVM architecture. The application's UI has the pages that the user interacts with while using the application.

View-model Proxies

The application UI layer has UI pages. A view-model is associated with each UI page. Because replicas of every view-model exist on the server-side, the view-models available on the client-side are their proxies. Because they represent on the client-side the current state of view-model objects on the server-side, there is no data processing on the client-side. The view-model proxies work as data binding entities for the UI pages and provide one- or two-way automating updates to the UI pages they are bound with. The view-model proxies state is tightly bound with the view-model on the server-side. As soon as the UI page is displayed on the phone, the view-model proxy is created, the API

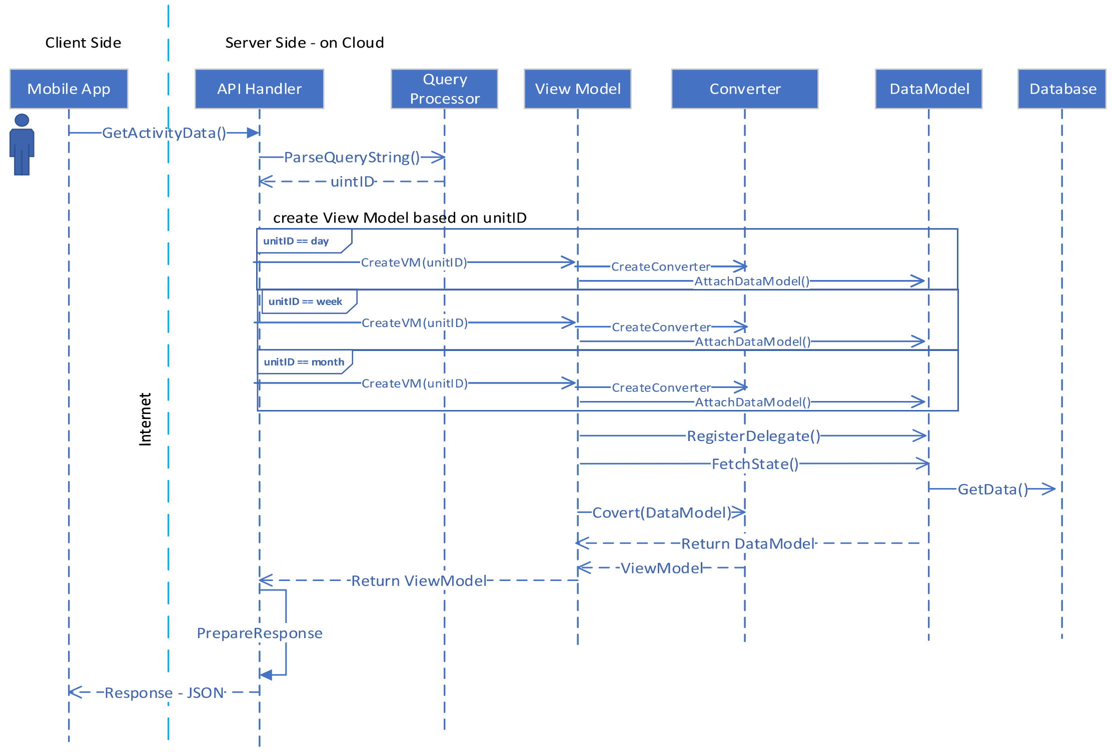


Fig. 4. RMVRVM execution on server-side.

call is made to get its state from the server, and the view (UI page) is updated using data binding.

JSON to View-model Mapper

The server-side sends the API response with an embedded JSON object. The JSON object's structure matches precisely the view-model proxy. Hence, the role of the JSON to view-model mapper is to deserialize the JSON into an object of the view-model. Notably, there are no heavy-duty conversions, filtering or any other kind of processing of the API response involved. In front-end programming languages like JavaScript, the JSON is a valid object. Hence, even deserialization may not always be required. The copying can be avoided using the move constructs in the programming languages, in which case the reference is copied, and the existing temporary object (JSON in memory, in this case) is rechristened as the view-model proxy object. Once the view-model proxy object's state is constructed, the view (UI page) gets updated automatically as it is tightly bound to it.

In the RMVRVM paradigm, the client-side design has become much more straightforward than in the MVVM paradigm. The key benefit of using the RMVRVM paradigm is a simple client-side design. The client-side design has become more straightforward as compared to the traditional MVVM paradigm due to the following reasons:

- The model objects are eliminated. Because the API response received on the client-side matches the state of its view-model, the model is not needed anymore.
- No converters are required for transforming models to view-models and vice-versa, avoiding expending the CPU for these transformations, thus saving battery
- No excess data is received on the client-side; therefore, no filtering is required of the data received in the REST API request, saving the battery.
- The number of objects to be created on the client-side is reduced due to eliminating the model layer and simplifying the mapping code. This reduces the CPU cycles and hence helps save the battery further.

As is visible in Fig. 3, client-side design in the RMVRVM architecture is very simple. This is desirable. A simple client-side design results in minimal utilization of CPU, memory and network, which reduces energy consumption, thus resulting in battery longevity. This is the most remarkable result of using the RMVRVM.

4.2. *Energy consumption on server-side*

The novel RMVRVM paradigm moves the complexity of task execution in an application to the server-side. This section discusses the details of energy consumption on the server-side for the application that employs the RMVRVM paradigm.

4.2.1. *RMVRVM paradigm — the complexity shift*

The authors propose using the novel RMVRVM paradigm instead of paradigms like MVVM in the UI-based applications connected to the cloud. The class of applications where the RMVRVM paradigm is better includes all user interface-heavy applications that run on various devices, including smartphones, laptops, smart watches etc. These applications are connected to the cloud for the to-and-fro data transfer to run the user scenarios. The RMVRVM paradigm advocates moving most of the business logic to the server-side, thus simplifying client-side implementation. The comparison of architectures of traditional MVVM and RMVRVM is shown in Fig. 5.

The comparison of architectures for the client-side of the application from Fig. 5 reveals that the complexity is markedly reduced in the RMVRVM paradigm. The dotted and grayed boxes in Fig. 5 indicate the portions of the traditional MVVM architecture eliminated in the RMVRVM paradigm. This is because the response objects of the API that the front-end uses in the implementation recommended in RMVRVM the paradigm contains the JSON objects just sufficient for the user interface. Hence, no response processing is required on the front end or client-side. This makes the client-side architecture in the RMVRVM paradigm much more straightforward than the traditional MVVM one. It is to be noted that there is no compromise on the application's capability. The user interface pages still have one-way or two-way

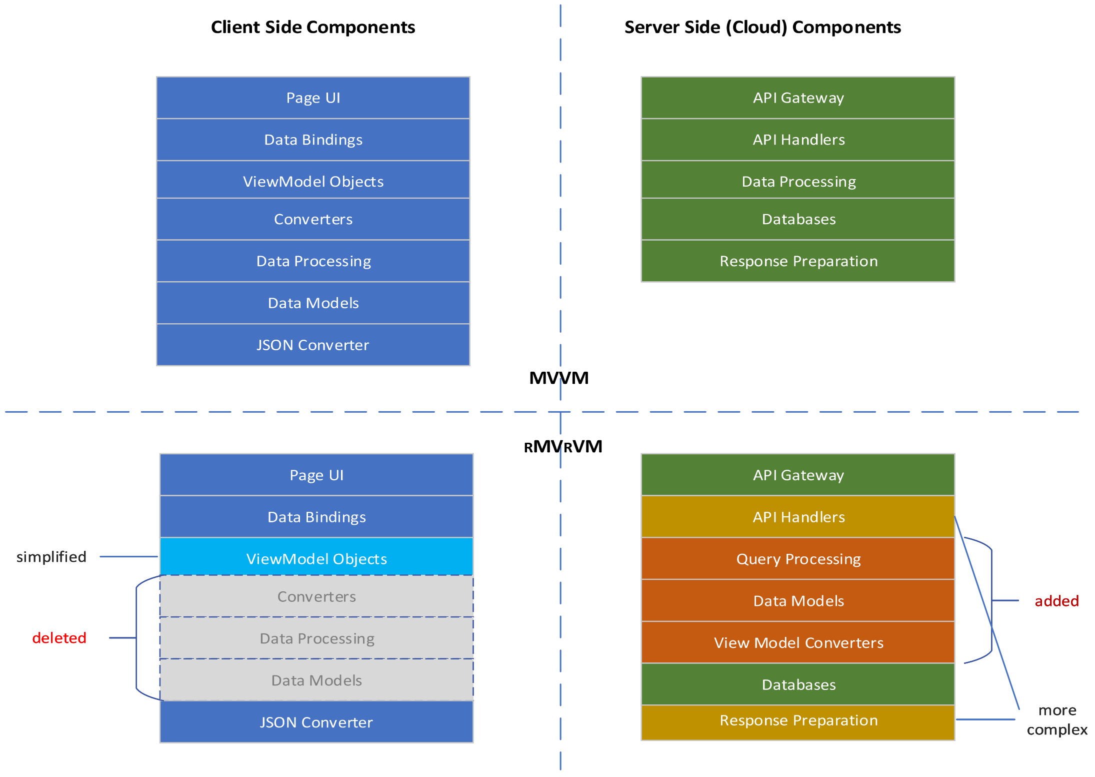


Fig. 5. Complexity shift in RMVRVM. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

data bindings with the underlying view models. The view models get updated directly with the API response instead of going through more steps in MVVM. In the RMVRVM paradigm, view models on the client-side are merely proxies of actual view models on the server-side.

The right-side of Fig. 5 compares the server-side architecture of traditional MVVM and RMVRVM paradigms. The server-side in MVVM is a typical design where the API layer receives the request from the client-side, and the rest of the layers may or may not be present, depending upon the complexity of the application. In the RMVRVM paradigm, the server-side architecture is significantly more complex. The red boxes mark the added complexity of this architecture. The observation is that the model and the view-model with their conversion objects have been moved to the server-side. The processing of API is additional work because the API is fine-grained and requires further extraction of the parameters. This extraction of parameters is also required in the MVVM design, but in the RMVRVM design, it is more pronounced due to a more significant number of more fine-grained APIs. The existence of remote view-model objects on the server-side requires that the converters are instantiated to convert from the data model objects to view-model objects. This conversion adds to the complexity as well. The response preparation is marginally more complex than the MVVM paradigm, too, because the response preparation in the RMVRVM paradigm is a fine-grained activity that requires serializing view-model objects of different pages into JSON, bringing in many more JSON templates for the conversion than MVVM which may be used on one or two JSON templates.

4.2.2. Discussion on energy savings in cloud data center

The server-side in the RMVRVM paradigm is running on the cloud — that is, in the data center of a cloud service provider. The application's server-side code can run either on a dedicated virtual machine, aka in IaaS mode or using the web API hosting service of the cloud service provider, aka in PaaS mode. The architecture is immune to the choice of IaaS or PaaS implementation for the application's server-side solution. Cloud data centers host numerous servers and run hundreds of applications simultaneously using technologies like virtualization, containerization, horizontal data partitions etc. Therefore, the empirical data about the energy consumed by a particular application running in the cloud cannot be obtained directly because the energy consumption in the data center is tracked at the equipment level. Because of this reason, the authors studied the literature on various techniques used in data centers to reduce energy consumption. Then, they added reasonable explanations to link these energy savings to the RMVRVM application to argue that running tasks on a data center is better than running the same tasks on disparate client devices.

Table 1
Composition of energy consumption in a cloud data center.

Component	% of total energy
IT equipment	50%
Cooling systems	40%
Other infrastructure	10%

The energy consumption of a cloud data center depends on parts like IT equipment, cooling systems and other infrastructure (Cheng et al., 2021), as shown in Table 1.

The IT equipment includes the hardware of computing systems like the servers, the network, the storage etc. The cooling systems include computer room air-conditioners (CRAC), compressors, fans, etc.; other infrastructure includes power supply, lighting systems, etc. The energy-saving efforts in a cloud data center are on the IT equipment and the cooling system. A measure of efficiency in energy consumption called (the Power Usage Effectiveness) PUE index is widely used by cloud service providers to publish the energy efficiency of their data centers. PUE = Total Energy Consumption/IT Equipment Energy Consumption. A PUE value closer to 1 is considered better for a data center.

In the RMVRVM paradigm, the server-side code runs on a virtual machine, consuming energy primarily in CPU, GPU, and memory. The energy consumption by all applications running on IT servers through virtual machines results in IT servers consuming energy in data centers.

Though inevitable, energy consumption can be reduced by employing appropriate technical solutions. As per the study conducted in Cheng et al. (2021), the technical solutions to reduce power consumption by IT servers can be divided into three categories: dynamic voltage and frequency scaling (DVFS), shutting down idle servers, and virtualization technology.

The energy consumption by CPU, GPU and memory when an application runs can be estimated (Bergman, 2020) using Thermal Design Power (TDP). Hardware manufacturers provide the TDP number to help users with their hardware design cooling systems. There is a correlation between the TDP number and the energy consumed by the component, though TDP is not directly a measure of energy consumption. If TDP is taken as an approximation of energy consumption for each component, then the energy consumed by the application will be given by,

$$P[\mathrm{kW}] = \frac{(c.Pc + Pr + g.Pg)}{1000} \tag{1}$$

$$E[\mathrm{kWh}] = P[\mathrm{kW}] * t, \tag{2}$$

where, c = number of CPUs, Pc = TDP of CPU, Pr = TDP of memory, g = number of GPUs, Pg = TDP of GPU, t = time

To account for the cooling employed in a data center, we need to multiply it by the PUE index to get the actual energy consumption of the application:

$$Energy\_Consumption = E[\mathrm{kWh}] * PUE \tag{3}$$

As of the current study of the available literature by the authors, the cloud service providers do not yet provide information about the CPU level energy consumption on a hardware server running underneath a virtual machine. The hardware server is the actual seat of energy consumption. Further study on calculating actual energy consumption by the application running on the cloud is planned as a future study. However, the authors provide the following reasons to argue that the proposed paradigm effectively saves the overall energy consumption of an application and, hence, helps reduce the software's carbon footprint.

- RMVRVM reduces the API load, i.e., the volume of data that needs to travel from the server to the client-side is reduced. Hence, a net reduction of energy consumed over the network can be expected.
- In Section 5, using experiments and case studies, the RMVRVM paradigm is shown to reduce the carbon footprint on the client-side. The server-side is usually a more controlled environment, and the server providers employ green energy practices at a data center level to reduce the overall carbon footprint. This could reduce the degree to which more energy is required because the computations are moved to the server-side by RMVRVM.
- A cumulative gain in energy saving is apparent because, due to the new paradigm, the energy is saved at every client device, which could be in millions. The move to the server-side consolidates the resources on the server and will be more energy efficient because of the data centers' better utilization of server-side hardware.
- When a task is run on the CPU of many mobile devices (MVVM) versus the same task run multiple times on a single web server (RMVRVM), the energy expended in the former will be more than the latter.
- Consolidating the application code that consumes energy to a single location by the paradigm opens up possibilities for further optimizations like memory caching, reducing database queries and reducing the overall energy consumed by memory, network and database servers. Such a caching solution is impossible when the application runs tasks on the client-side.

4.3. Adopting RMVRVM

In this section, we discuss the details related to the adoption of RMVRVM by practitioners. First, we provide a framework or guidelines to migrate an application from MVVM to RMVRVM. In the next part, we discuss challenges that the practitioners will likely face while implementing the RMVRVM for their applications. And then, in the end, we also discuss whether RMVRVM could be a complete replacement of the MVVM paradigm in UI-centric applications regardless of the device for which they are built to run.

4.3.1. Migrating MVVM to RMVRVM

Numerous applications are using the MVVM paradigm. With the introduction of the RMVRVM paradigm, practitioners may want to migrate these applications to use the RMVRVM. However, adopting a new design pattern that significantly alters the application's architecture is always challenging. To address this concern and encourage the adoption of the RMVRVM paradigm, we provide the framework, as illustrated in Fig. 6, that could help practitioners migrate their applications to RMVRVM. This framework aims to provide ready guidance to minimize manual work while migrating.

Step 1 — Create Remote View Models

In the RMVRVM, the view models of the front-end UI pages of the application exist on the server, which are called remote view models. However, they are likely a replica of the view models in the MVVM. Hence, step one advocates reusing existing view models as remote view models on the server-side. The programming language and framework used in the front end could be different from the back end. For example, the front end could use JavaScript, and the back end could use Java or C#. But it is also possible that the technology stacks are the same for both the back and front ends. For example, the front end could use JavaScript, and the back end could use Node.js — a JavaScript-based framework. Or, the front end could be using the .NET MAUI framework in C#, and the back end could also be in C#. Step 1 of the framework handles both scenarios. If the technology is the same, the view models can be moved to the back-end code and reused directly. If the technology stack differs, tools are available (Bergman, 2020) to convert entities from one language to another.

Step 2 — Define API and Prepare Response

In the RMVRVM paradigm, the API definition should provide the view models in response. Therefore, the migration framework advocates one API per view model. Step 2 of the framework uses JSON.stringify() function present in most JavaScript frameworks to convert view models to the JSON format. These JSON representations of the view models are the ones that the API will fill with data and return as the response. Therefore, these JSON view-model entities can be used in the response preparation code of the API controllers in the back end. The definitions for each remote view-model API from these JSON entities can also be created. The view models bound to their UI pages through one-way binding in MVVM will translate into GET API definition if the binding is from the view-model to the UI element. However, the one-way binding from the UI element to the view-model will translate into a POST API definition. In the case of two-way data binding, GET and POST API definitions will be created on the server-side for those view models separately.

As steps 1 and 2 show, the migration framework uses the view models in MVVM as the pivot to drive the implementation of the RMVRVM on the server-side.

Step 3 — Client-side clean up

Step 3 of the migration framework involves cleaning up and getting the code and application ready to reap the benefits of the RMVRVM adoption. The data processors implemented on the client-side were specific to the server-side's API response. These data processors can be deleted after step 2 creates the new API definitions. These may not be reusable on the server-side because, in all likelihood, there would be better ways to query the data from the databases to get

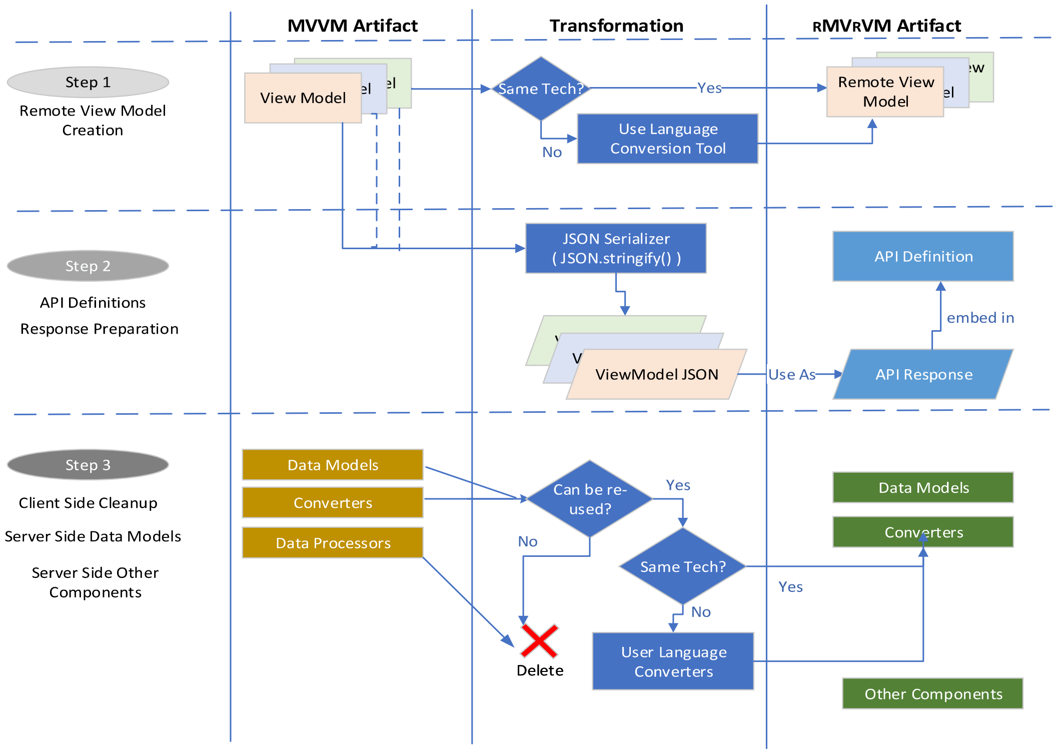


Fig. 6. Framework for migrating an MVVM application to RMVRVM.

the data to the data models. For migrating the converters and data model objects, practitioners are advised to see if they are reusable on the server-side. If they are, we can move them to the server-side directly or through language converters, like in step 1. Once these conversions are complete, the other components could be required for server implementation, depending on the application's business logic.

The migration is complete after using these three steps. At this point, the authors recommend that the system be put through a test cycle to validate the accuracy of the new system. Once the accuracy is established, practitioners can take advantage of the fact that, post the migration, all data and logic are moved to a central place in the cloud. For example, they can use data caching to reduce the system's response time.

The authors used this framework to migrate the open-source application from MVVM to RMVRVM. The details of this migration are discussed in Section 5.2.

4.3.2. Common challenges to adoption of RMVRVM

This subsection discusses common challenges practitioners could face while adopting the RMVRVM. For each challenge discussed in this subsection, we first define the challenge, elaborate on its technical detail and then propose possible solutions to handle the challenge.

Challenge 1: How to handle 3rd party API which is not under the application author's direct control?

The client-side of the application might be invoking 3rd party API directly. These APIs are not under the direct control of the applications' author. Adopting RMVRVM requires altering the API definitions. So, the challenge is how practitioners can adopt RMVRVM in the presence of 3rd party API.

Discussion & Solutions

Generally, the client-side of applications calls both 3rd party API and its internal REST API. In this case, the solution would be to call the 3rd party API from the server-side and provide a facade API for the client application to call. Once this change is done, the client-side no longer calls the 3rd party API directly. It calls its internal API, and that API calls the 3rd party API. This is a better design because now the client-side does not know which 3rd party API the server-side is calling, thus removing the tight coupling it had with that 3rd party API. Now that the client-side dependency on the 3rd party API is gone, we can refactor the internal REST API to conform to the RMVRVM paradigm.

There could be applications that do not have the server-side API at all. They operate purely using third-party API. For example, a developer could write a smartphone weather application that uses third-party weather services only and has no internal API. In such cases, the RMVRVM paradigm is not applicable because it requires a server-side application running on the cloud.

Challenge 2: A mobile application could have tens or even hundreds of UI pages. Many of these UI pages dynamically refresh. Is adopting RMVRVM for such applications possible?

The applications like e-commerce, travel planning, e-library etc., on smartphones have many UI pages. These UI pages refresh their data dynamically. If the view-models of all these pages are moved to the server-side, it will complicate the API. How can the practitioners adopt the RMVRVM paradigm for such applications?

Discussion & Solutions

There are two issues in this challenge. First, how to handle the complication in API arising out of all view-models moving to the server. Second, how to manage moving so many view-models to the server-side. The solution to the first issue is careful nomenclature and definition of the API. The API should be named against the UI page it serves. This way, there will be an order in the API definitions. The API should be aggregated in cloud services like API Gateways that help manage the complexity of the API by providing API aggregation and other facilities. The second issue is a project management issue. The transition to RMVRVM is recommended to be gradual. Having RMVRVM and MVVM in one application is okay. Therefore, the recommendation is that during the development of the application features, each sprint adopts RMVRVM to a fixed number of UI pages. At the end of the sprint, the RMVRVM-compliant pages can be moved to production. This iterative approach can move all UI pages to RMVRVM.

In this iterative approach, it is recommended that the API of UI pages should be kept separate from the current API. The API Gateway's API aggregation facilities can be used to have the new API internally call the existing API if required.

Challenge 3: How should the complexity moved to server-side by the RMVRVM be managed?

The RMVRVM shifts the complexity of client-side business logic to server-side. The server-side may already have complex code catering to the API it provides for the client-side. Now, the client-side complexity gets mixed up with server-side. The challenge, therefore, is to find ways to manage this complexity.

Discussion & Solutions

As the architecture of RMVRVM in Fig. 3 clearly defines the roles of each component, it is recommended that the practitioners follow this architecture to handle the complexity on the server-side. Besides, the practitioners should carefully consider the view-models state and the model objects. There could be opportunities to use mechanisms like caching to speed up the response further and avoid unnecessary calls to the databases to get the data into the model and view-model objects.

Bringing the complexity to server-side is advantageous for the developers. The server-side is deployed on the cloud, and the application's developers control it, but the client-side is deployed on the devices held by the end users. Therefore, it is easier for developers to update the application components on the server-side without shipping an upgrade to end-users' individual smartphones. The server-side changes can be developed and deployed rapidly using agile software practices to better the functioning of the code. While adopting the RMVRVM paradigm comes at the cost of moving complexity to the server-side, the authors believe this liability can be advantageous.

Challenge 4: During the adoption of RMVRVM, how can the practitioners identify unused and excess data on the client-side?

The authors of this paper claim that the RMVRVM paradigm eliminates the excess data received by the client-side through the traditional REST API. The challenge, therefore, is how practitioners can identify the access data that should be eliminated on the client-side.

Discussion & Solutions

The excess data is present on the client-side for two reasons. First, the traditional API defined for desktop or web applications generally sends collections of objects and expects the clients to process them as needed. Second, the model objects on the client-side may have properties that are never used by the UI pages of the applications. Adopting RMVRVM moves the models and view-models to server-side. Doing so will require updating the response of the existing API or creating a new API. So, the solution to this challenge is that adopting RMVRVM will automatically eliminate excess data. Practitioners need not identify excess data themselves. The guidance here is that the practitioners must ensure that the response of the API matches the view-model in RMVRVM. Following this guidance will avoid sending excess data to the client-side.

Challenge 5: Modifying the REST API is a breaking change; hence, the cost of adopting RMVRVM is high.

The RMVRVM advocates refactoring the REST API into a more granular API that handles each view-model separately. The REST API in production is used not only by the mobile application but also by web and desktop applications. If the REST API is updated specifically for mobile phones, it will break the other applications. Hence, changing REST API is expensive because all other applications must change. How can this be handled more efficiently?

Discussion & Solutions

The refactoring can be done in a way that does not break the current API. For example, additional query parameters can be introduced with default values, and the server-side code can execute refactored code under RMVRVM if the value of the additional query parameter is received from the client-side. Otherwise, it can continue to serve the MVVM response.

Challenge 6: The UI frameworks, like Angular and React, force unnecessary server calls from the client.

The RMVRVM paradigm advocates the reduction of unnecessary calls to the server-side to avoid getting unnecessary data. However, most popular UI frameworks, like Angular and React, are used in many client-side applications for unnecessary server calls because the client-side script code comes packaged in modules to the client-side. There is no way for the developer to unbundle the package on the server-side when RMVRVM is being applied.

Discussion & Solutions

This challenge can be addressed using a recent server-side rendering technique. In server-side rendering, the UI frameworks allow execution of the code on the server-side to create simpler web page objects for browsers to display. Once the code uses this feature, the RMVRVM can be applied because the server-side rendering removes the unnecessary calls from the client-side.

4.3.3. Can the RMVRVM completely replace MVVM?

The RMVRVM is superior to MVVM in the case of UI-centric applications running on energy-constrained devices like smartphones. As the data models and business logic are moved to a central place on the cloud, it provides opportunities to optimize the code further using mechanisms like caching, query optimizations, and the reuse of components. Can the RMVRVM completely replace MVVM in all kinds of UI-centric applications like desktop clients and web applications (i.e., applications running in the browser)?

The RMVRVM paradigm comes with its complications. Here are some cons of using the RMVRVM paradigm.

- It makes the API definition on the server-side fine-grained and, therefore, verbose to enable remote view-model serialization into the API response. This can make the server-side code complex.
- As Section 4.2.1 illustrates, when the RMVRVM is applied, the complexity of the application shifts to the cloud. This could result in a need to provide additional computing and storage resources, resulting in increased operating costs for the application.
- In some cases, the application's author does not own the API that the application uses on the client-side. It could be that all APIs used are from 3rd party service providers. Applying the RMVRVM paradigm is impossible.

Therefore, the RMVRVM cannot wholly replace the MVVM design paradigm. This paper discusses specific scenarios where the RMVRVM paradigm brings significant benefits. But MVVM is more useful in cases where there is no constraint on the resources on the client-side device where the application is running. For example, a user-centric desktop application connected to the cloud does not need to apply the RMVRVM paradigm because there is no shortage of resources — the power or energy is unlimited. After all, the desktop is plugged into a power supply, the memory and storage expansion is plug and play, and CPU and other upgrades are also possible to enhance the computing power of the desktop. Applying RMVRVM in such a system will underutilize the resources. Using the MVVM paradigm instead, the desktop machine's capabilities can be utilized better by processing tasks locally, pre-populating the caches and data model collections at the start of the application to avoid frequent network calls, etc. In this case, for example, MVVM is a better choice than RMVRVM.

Energy consumed by the network vs. the CPU

The RMVRVM expends more energy on the network than the MVVM but saves much more energy in CPU execution on smartphones than the MVVM because of the complexity shift to the server-side. This is evident in the experiment conducted in the next section. The experiment results show that despite the penalty of network traffic, the RMVRVM paradigm saves substantial energy on smartphones (see Section 5.1.4). Extrapolating this observation further, it can be seen that the RMVRVM paradigm will not be suitable for applications that use the network substantially more than the CPU. For example, a background application that updates a database installed locally on the smartphone. Such a class of applications does not use any UI framework anyway.

5. RMVRVM — Evaluation

In the previous work (Singh, 2022a), we conducted experiments and applied the RMVRVM paradigm to two industrial case studies to validate its efficacy. In this paper, we expand on testing the efficacy of the RMVRVM paradigm in two ways. First, an experimental mobile application is built (Singh, 2022b) that executes CPU-bound tasks on the phone. It is run on iPhone and three Android phones using 4G and Wi-Fi to simulate the running of real-world applications. Second, we select an open-source MVVM-based application and apply a migration framework to make it RMVRVM-compliant. We record its energy consumption and response time before and after migration and compare the results.

5.1. Experiments

A smartphone application was developed in C# using the Xamarin platform for conducting these experiments. Xamarin cross-platform application development technology helped us deploy the application to iOS without significant changes in the source code. This application simulates the workload on the smartphone by executing a CPU-bound task repeatedly to expend energy. The application tracks the usage of the smartphone's battery through the API provided by the OS SDK. The application connects to a back-end service. It gives the option to run the same CPU-bound task on the cloud. Thus, it can simulate both cases — (1) a smartphone application that uses MVVM and (2) a smartphone application that adopts RMVRVM. Upon the end of the experiment, the application uploads the readings taken to the cloud in CSV format that can be used to analyze the results.

5.1.1. Source code

The author's public GitHub repository (Singh, 2022b) has the simulator application's source code, the back-end application source code, and the APK used to deploy the application on the three Android phones in the experiments.

The simulation of workload on CPU was performed using *tan(arctan (tan(arctan...* load function executed repeatedly. This method invokes CPU execution. The PDP-11 microprocessor was tested using the CPU soak test that used this method. The iterations to execute the method were set to *Int32.MaxValue* (Reference, 2022), which is 2,147,483,647 in the .NET framework. A text field is provided in the application's UI to configure the number of iterations to a lower value during the experiment. This text field takes a number, which is called reverse-intensity value. The reverse intensity value divides the maximum iteration value to bring it to a lower number. This helps calibrate the application according to the hardware profile of the device. This is needed because the application creates a new thread every second and executes the CPU-bound method on the thread. There is a hardware limitation on the number of threads that can be created on a smartphone. The reverse intensity value can reduce the number of iterations to allow the application to keep the number of threads within the hardware limit.

The application code keeps the UI thread free of task executions. This allows the application to stay interactive and receive user input while executing the tasks in the background. This allows the user to end the experiment at will.

Table 2
Configurations of smartphones used in the experiment.

Phone	Battery	CPU/RAM
OnePlus A6000	3300 mAh	Snapdragon 845/8 GB
Samsung Galaxy F62	7000 mAh	2.6 GHz/6 GB
Asus 6Z	5000 mAh	2.8 GHz/6 GB
iPhone SE	1624 mAh	A13 Bionic/4 GB

5.1.2. Using the simulator application

The application UI has radio buttons to choose MVVM or RMVRVM. If the user chooses MVVM, it will run the CPU-bound task on the smartphone. Otherwise, it runs on the cloud. When the application starts, the user can start the experiment by tapping the Start Experiment button. Note that the application connects with the back-end service. Hence, to complete the experiment successfully, the back-end service on the cloud should be running. After starting the experiment, the application displays the status of the tasks that are getting executed in the background. It also displays the battery consumption so far and the time since the start of the experiment (Fig. 7). For the simple implementation, the application shows the status of the last 50 tasks on the UI. The user can stop the experiment using the Stop Experiment button. After the experiment is stopped, the application sends the battery consumption data recorded since the experiment's start to the cloud. When the user opts for MVVM, the task execution happens on the smartphone. There are no network requests sent or received during the MVVM experiment. When the user chooses RMVRVM, the tasks are executed on the cloud. There is network traffic in this case — the status of tasks is queried by the client-side using the API. This simulates the actual scenario where the application executes tasks in the cloud, and the user sees the results in the smartphone UI.

5.1.3. Conducting the experiment

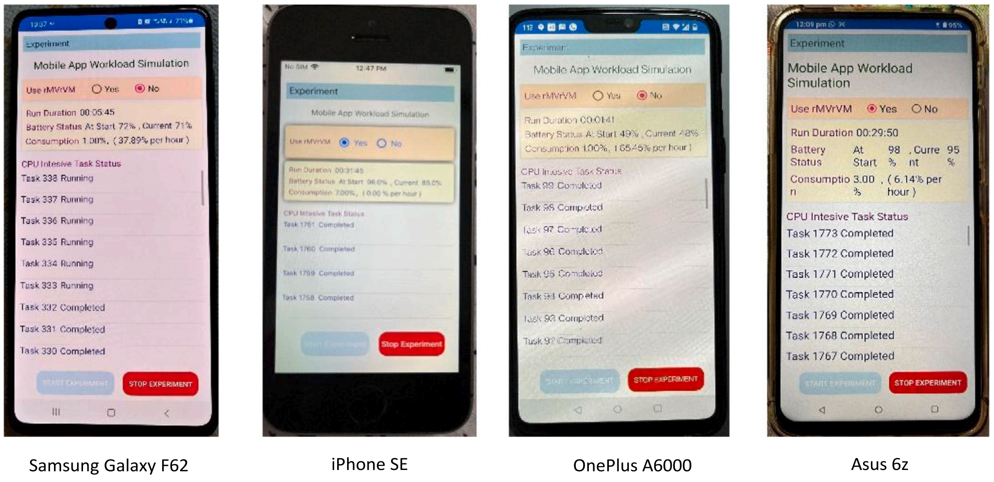
The experiment was run on three different Android phones and one iPhone. The configurations of these phones are shown in Table 2.

To experiment, the display timeout was set to never to keep the display enabled all through the experiment. The adaptive brightness feature was turned off on all phones. The lock screen and screen savers were turned off too. The phones' battery was charged high to ensure it lasted for both MVVM and RMVRVM options chosen during the experiment. It was also ensured that no other applications were running on the phone. This exercise aims to create a similar environment for the experiment. However, it is essential to note that the experiment does not include comparing the energy consumption across the phones. It compares MVVM and RMVRVM on a single device. Therefore, the experiment would compare MVVM and RMVRVM for four phones. The experiment for iPhone SE and OnePlus 6 was conducted for both 4G and Wi-Fi.

5.1.4. Experiment results, data analysis, and discussion

Fig. 8 shows the experiment results for all four phones. The plots are drawn from the report of the simulator application sent to the cloud. The data was imported into spreadsheet software and analyzed. The plots were then created on spreadsheet software. The plots validate the efficacy of the RMVRVM paradigm. The phone battery consumption is reduced substantially when the application runs in the RMVRVM simulation.

Among the three Android phones, when on Wi-Fi, the Samsung Galaxy F62 had the highest battery saving of 66% using the RMVRVM paradigm, followed by the OnePlus 6 phone (60%) and Asus (50%). When using 4G, the OnePlus 6 phone consumed 70% lesser battery. The iPhone saved 53% energy when on Wi-Fi, but it saved a whopping 82% when 4G was used! The graph's slope measures the battery consumption rate per unit of time. The plots' battery consumption rate can be observed from the graphs to be linear because the experiment loads the CPU using code that runs in a loop, i.e., with a time complexity of



Experiment
Mobile App Workload Simulation
Use rMVrVM Yes [✓ No]
Run Duration 00:05:45
Battery Status At Start 72% , Current 71%
Consumption 1.00%, (37.89% per hour)
CPU Intesive Task Status
Task 338 Running
Task 337 Running
Task 336 Running
Task 335 Running
Task 334 Running
Task 333 Running
Task 332 Completed
Task 331 Completed
Task 330 Completed

Experiment
Mobile App Workload Simulation
Use rMVrVM [✓ Yes] No
Run Duration 00:31:45
Battery Status At Start 96.0% , Current 89.0%
Consumption 7.00%, (0.00 % per hour)
CPU Intesive Task Status
Task 1761 Completed
Task 1760 Completed
Task 1759 Completed
Task 1758 Completed

Experiment
Mobile App Workload Simulation
Use rMVrVM Yes [✓ No]
Run Duration 00:01:41
Battery Status At Start 49% , Current 48%
Consumption 1.00%, (65.45% per hour)
CPU Intesive Task Status
Task 99 Completed
Task 98 Completed
Task 97 Completed
Task 96 Completed
Task 95 Completed
Task 94 Completed
Task 93 Completed
Task 92 Completed

Experiment
Mobile App Workload Simulation
Use rMVrVM [✓ Yes] No
Run Duration 00:29:50
Battery Status At Start 98 % , Current 95 %
Consumption 3.00 % , (6.14% per hour)
CPU Intesive Task Status
Task 1773 Completed
Task 1772 Completed
Task 1771 Completed
Task 1770 Completed
Task 1769 Completed
Task 1768 Completed
Task 1767 Completed

Fig. 7. Application user interface.

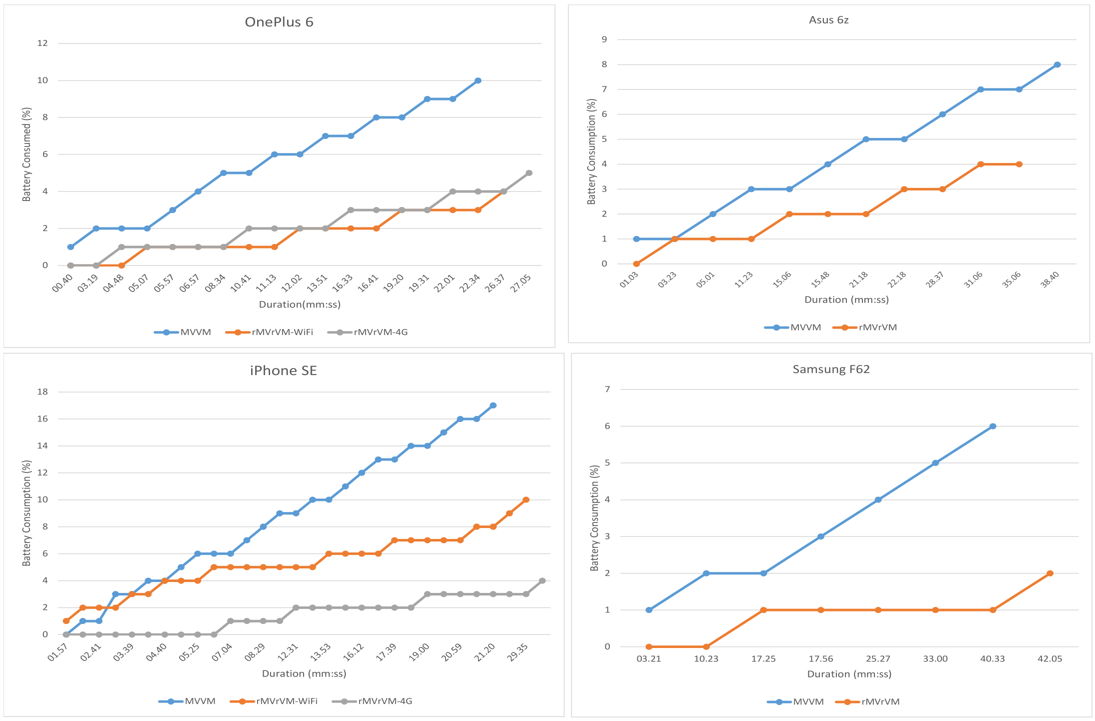


Fig. 8. Battery consumption experiment results.

O(N). On average, the battery consumption rate for MVVM is 3.5 times that of the RMVRVM paradigm.

The RMVRVM paradigm has the penalty of doing network calls during the experiment, which is not the case with MVVM. The RMVRVM brings task status from the cloud to the client. Despite this penalty, there are substantial savings in battery consumption in the RMVRVM mode of the simulator application. This gives us a convincing argument that CPU-intensive tasks consume disproportionately large batteries compared to the battery consumed by the network traffic.

5.2. Open-source case study

To further validate the applicability of the RMVRVM in an actual application, open source was searched for an application code that used the MVVM paradigm on the client/mobile side and had a back-end implemented to respond to the requests from the client-side. After considerable research and evaluation of different applications, the authors chose the Restaurant App to evaluate the new paradigm. This application has been written to demonstrate how MVVM design can be used on the mobile front-end and the microservices architecture can be applied on the server-side using modern technology mechanisms like containers, API gateways, proper authentication etc. The client-side or mobile application is authored in C# and uses the Xamarin framework for the UI experience. The client-side of this open-source application is the primary area of interest in this case study.

The case study was conducted in four phases. Phase 1 involved preparing the code; in Phase 2, the application was run to collect data with MVVM already in place in the original code; in Phase 3, RMVRVM was applied, and then the application was run in the same conditions as Phase 2, and the data was recorded. The analysis of the data and the comparison were done in Phase 4 of this case study. The details of the case study are now discussed.

5.2.1. Phase 1 — Preparation

The original repository on GitHub[1] was forked to the author's own GitHub account.[2] This is required to make the necessary changes independently. The forked open-source repository on the author's GitHub account has public access. All the changes done by the authors in this case study are available for review in the forked repository. The original code uses mock data instead of calling an API for debugging. This code was modified to call an actual API on the cloud. This API, GetMenu, was implemented by the authors on the server-side. For MVVM, the API returned a JSON of a restaurant's menu picked from open-source.[3] On the client-side the converter code was implemented to extract out the data that the Foods page needs to display. To instrument the application, access to the Battery API of Xamarin is provided in Xamarin. Essentials NuGet package was needed. This required that the application be upgraded to Xamarin.Forms 5.0. The application was upgraded, and an instrumentation code was added. The only UI change was to add a button to refresh the menu list and another button to upload the instrumentation data collected during the running of the application when the upload button is tapped. One tap on the Refresh button means executing one cycle of getting the menu data, which involves calling the GetMenu API, doing the necessary filtering and updating the model, and then updating the view-model. The count of Refresh button taps was kept under the track and uploaded. This count enabled us to measure the application's response time in both scenarios, as explained in the data analysis section.

5.2.2. Phase 2 — Execution with MVVM

The original code of the application uses MVVM. When the application starts, it shows the Foods page, which lists food items. This page might be viewed often during application usage in the real world. In this case study, this was simulated by the first author by clicking the Refresh button. The method of doing this continuous refresh was the following. When the list of food items is displayed, the author presses the refresh button again. The list gets refreshed due to the click. Again, the button was pressed. This manual refresh was done for about 78 min to obtain the required data.

Fig. 9 shows a UML sequence diagram of the execution flow when the food menu is displayed on the application. The Foods View represents the View object that displays the page to the user. As it is about to be displayed due to the start of the application or tap on the refresh button, its OnLoading() function is called by the operating system and in this OnLoading() function, it calls the LoadFoods() function on its view-model object. This view-model object calls the API handler's GetFoods() function. The GetFoods() function is responsible for calling the GetMenu REST API. Once the response is received, the GetFoods() function processes this response by calling the ParseRestaurantMenu() function. The API response is parsed and filtered in this function to create a list of FoodDto objects. FoodDto is the model object for FoodsViewModel.

The collection of the FoodDto object is sent back to the FoodsViewModel, which then uses the mapper or converter object to create a collection of the FoodViewModel list, which is data bound to the Foods view. Due to this data binding, the UI of the Foods page gets refreshed automatically because the source of the underlying items got updated. This process is repeated every time the refresh button is pressed.

In Fig. 9, the orange dots or small circles identify processing hotspots. These are the steps where the CPU on the mobile phone will work to process the data and consume the battery. At the end of the usage of the application, the upload button is tapped. This triggers the process of uploading the instrumented data to the cloud. The instrumented data include the duration and the battery drainage by percentage. It also includes the count of times the Refresh button was pressed.

5.2.3. Phase 3 — Execution with RMVRVM

The RMVRVM paradigm requires appropriate changes on both the client and server-sides. In this phase, the first task was to make these code changes to make the open-source application implement the RMVRVM paradigm. Then, the next task was to conduct the experiment again and gather the instrumented data.

Code changes for RMVRVM

The code changes were done on both the client and server-sides to implement RMVRVM in the open-source application. The RMVRVM paradigm requires that the model and view-model objects are shifted to the server-side, and all processing should be done there. The same was implemented, and the ParseRestaurantMenu() function was moved to the server-side. In this case, since no other processing was required, there was no need to create a list of actual view-model objects and then serialize them to JSON. The result returned by ParseRestaurantMenu() represents the view-model and hence is directly persisted to JSON as it is. This JSON is then returned to the client.

On the client-side, the code became much more straightforward than expected when the RMVRVM paradigm was applied. There was no processing of the API response required because the response directly matched what the view-model needed. So, on the client-side the code that did the process on the incoming response from the API call was reduced from the following:

```
JObject data = JObject.Parse(json);
List<FoodDto> menu = new List<FoodDto>();
foreach (var item in data["categorys"]) {
    foreach (var menuItem in item["menu-items"]) {
        var foodItem = new FoodDto() {
            Id = menuItem["id"].Value<int>(),
            Name = menuItem["name"].Value<string>(),
            Description = menuItem["description"].Value<string>(),
            Price = menuItem["sub-items"][0]["price"].Value<Decimal>(),
            Cuisine = menuItem["sub-items"][0]["cuisine_name"].Value
            <string>() };
        foodItem.Price /= (decimal)70.0;
        menu.Add(foodItem); }
}
```

to just a single line of code:

```
var menu =
JsonConvert.DeserializeObject<List<FoodDto> >(await resp.Content.
ReadAsStringAsync());
```

The mapper was eliminated because the code did not require any transformation. A lightweight proxy of view-model created the collection directly from the response and sent the list back to the Foods view, which displayed the list. On the UI side, the refresh button was updated to show the number of times the refresh button has been clicked. The idea was to stop the execution once the refresh was done the same number of times as in the MVVM case.

Execution of the application with RMVRVM

Fig. 10 shows the UML sequence diagram of the execution flow when the application loads the Foods view either during start or due to tapping on the Refresh button. The client-side code has become much simpler because the processing hot spots have either been moved to the server-side or eliminated. The model and mapper objects on the client-side are eliminated, and the API handler prepares the collection of FoodViewModel objects for the Foods view to display. The API handler

[1] https://github.com/chayxana/Restaurant-App.

[2] https://github.com/LavneetSingh/Restaurant-App.

[3] https://github.com/terrenjpeterson/caloriecounter/blob/master/src/data/foods.json.

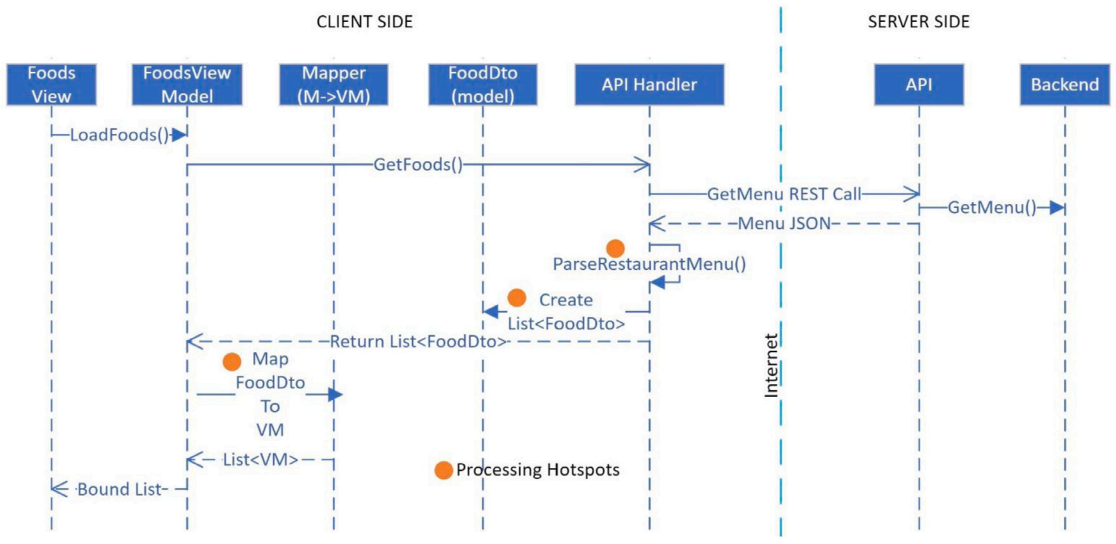


Fig. 9. MVVM execution in the open-source application.

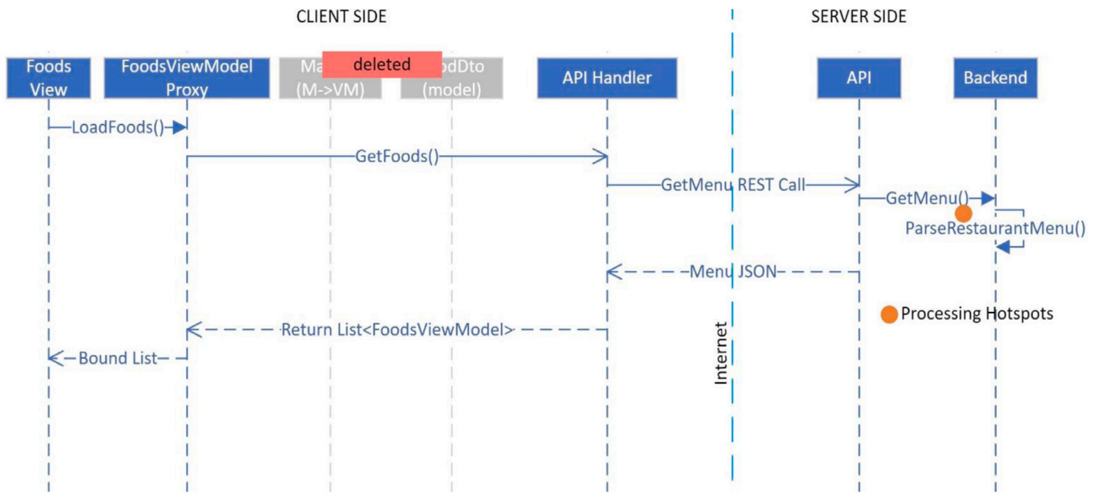


Fig. 10. RMVRVM execution in the open-source application.

can do that easily because the response that it receives from the server-side does not require any processing. After all, it is a direct fit to the view-model.

The application was executed following the mechanism of MVVM. The Refresh button was clicked every time the list was loaded. Notably, the author could see the Foods page refreshed much quicker. That is why a similar number of taps to refresh the page were finished much earlier than in the MVVM case.

5.2.4. Analysis, comparison, and conclusion

The instrumentation done in the application for both cases collected the data whenever there was a drop in the battery percentage. At that instance, it collected a tuple of duration since the application started and the battery percentage. The refresh count variable tracked the number of Refresh button presses. Notably, every press of the Refresh button would not cause a drop in the battery percentage. The refresh count represents completing one task, an end-to-end cycle to refresh the food menu on the UI.

For the MVVM, the data was collected for about 78 min, comprising 1246 refresh button taps. We can consider a tap a task; therefore, 1246 tasks were completed in 78 min. It is to be noted that the button was tapped immediately after the list could refresh, not before. For RMVRVM, the data was collected for 1268 taps in about 42 min.

Fig. 11(a) shows plots of battery consumption in completing 1246 units of tasks by both paradigms. It clearly shows that after the RMVRVM paradigm is applied, the application becomes more efficient and consumes less battery, in this case, 42% less. Fig. 11(b) shows the average response time calculated for completing the 1246 units of tasks completed in about 78 min. The average response time when the application uses RMVRVM is about 2 s compared to more than 3.6 s when the application uses MVVM. The response time is 45% better for the RMVRVM case. The data collected during this exercise and the data analysis are available for review in the code repository under the analysis folder.

5.2.5. Data sanctity and error margins

It should be noted that data collection involves manual human effort. The response time includes the time it takes to tap the button after the eye observes that the list is refreshed. To remove any bias in the two efforts, the Refresh button was tapped many times (1200+) for each case to balance out the latency or incoherence in the manual tap actions over time. The author also visually observed during this exercise that the application became much faster in loading the Foods page after applying RMVRVM. It should also be mentioned that the response time could be measured more accurately by instrumenting the application further. Still, since the difference between the response times is large,

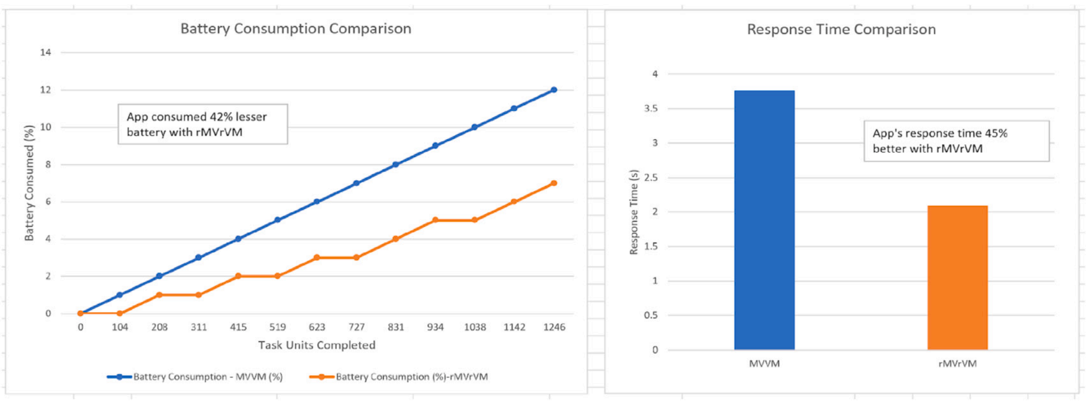


Fig. 11. Open-source application — comparisons, (a) Battery consumption (b) Response time.

the authors think it is sufficient to prove the efficacy of the proposed paradigm.

This case study helps establish that the proposed paradigm can be applied in real-world applications. The application will likely improve its response time and significantly reduce battery consumption.

6. Conclusion & future work

This paper takes forward substantially the work done by the authors in Singh (2022a). We provide further details to the novel RMVRVM paradigm proposed in Singh (2022a) to replace the traditional MVVM paradigm for energy-constrained devices like smartphones. We enhanced the experiments to include the iOS platform using the iPhone, experimented on both 4G and Wi-Fi, and added two more Android devices into the mix. We propose a migration framework to migrate MVVM-based applications to the RMVRVM. We apply the migration framework to an open-source application and migrate it successfully. This application's energy consumption and reaction time before and after the migration are presented, proving the RMVRVM paradigm's efficacy in saving energy and reducing response time. The challenges the practitioners could face while applying the RMVRVM paradigm are enhanced and include more scenarios.

Further, this paper discusses the server-side energy consumption by the RMVRVM paradigm, describing the complexity shift to the server-side. We discuss and argue that this complexity shift to the cloud is desirable because it allows us to use tactics like caching the data and taking advantage of the energy-saving practices adopted in the cloud data centres. We argue that it is reasonable to conclude that the overall energy consumed in the proposed paradigm will reduce the carbon footprint significantly due to reduced energy loss on billions of devices when considered together. In the future, the authors plan to experiment with the energy consumed by the cloud infrastructure and link it to the proposed paradigm.

CRediT authorship contribution statement

Lavneet Singh: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Saurabh Tiwari:** Methodology, Writing – review & editing, Project administration, Supervision. **Sanjay Srivastava:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Akherfi, K., Gerndt, M., Harroud, H., 2018. Mobile cloud computing for computation offloading: Issues and challenges. Appl. Comput. Inform. 14, 1–16.

Alshurafa, N., Eastwood, J.A., Nyamathi, S., Liu, J.J., Xu, W., Ghasemzadeh, H., Pourhomayoun, M., Sarrafzadeh, M., 2014. Improving compliance in remote healthcare systems through smartphone battery optimization. IEEE J. Biomed. Health Inf. 19, 57–63.

Android Jetpack, A., 2022. Data binding in andriod. https://developer.android.com/topic/libraries/data-binding. Accessed: 2023-04-25.

Bergman, S., 2020. How to measure energy consumption of your backend service. https://devblogs.microsoft.com/sustainable-software/how-to-measure-the-power-consumption-of-your-backend-service. Accessed: 2023-04-25.

Bolla, R., Khan, R., Parra, X., Repetto, M., 2014. Improving smartphones battery life by reducing energy waste of background applications. In: 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies. IEEE, pp. 123–130.

Carroll, A., Heiser, G., et al., 2010. An analysis of power consumption in a smartphone. In: USENIX Annual Technical Conference. Boston, MA, pp. 21–21.

Chen, E., Ogata, S., Horikawa, K., 2012. Offloading android applications to the cloud without customizing android. In: 2012 IEEE International Conference on Pervasive Computing and Communications Workshops. IEEE, pp. 788–793.

Cheng, H., Liu, B., Lin, W., Ma, Z., Li, K., Hsu, C.H., 2021. A survey of energy-saving technologies in cloud data centers. J. Supercomput. 77, 13385–13420.

Gupta, M., Koc, A.T., Vannithamby, R., 2011. Analyzing mobile applications and power consumption on smartphone over lte network. In: 2011 International Conference on Energy Aware Computing. IEEE, pp. 1–4.

IBM, 2022. What is a rest api. https://www.ibm.com/cloud/learn/rest-apis. Accessed: 2023-04-25.

Kim, M.W., Yun, D.G., Lee, J.M., Choi, S.G., 2012. Battery life time extension method using selective data reception on smartphone. In: The International Conference on Information Network 2012. IEEE, pp. 468–471.

Kouraklis, J., 2016. MVVM in Delphi: Architecting and Building Model View ViewModel Applications. APress, Berkeley, CA.

Pramanik, P.K.D., Sinhababu, N., Mukherjee, B., Padmanaban, S., Maity, A., Upadhyaya, B.K., Holm-Nielsen, J.B., Choudhury, P., 2019. Power consumption analysis, measurement, management, and issues: A state-of-the-art review of smartphone battery and energy usage. IEEE Access 7, 182113–182172.

Ragona, C., Granelli, F., Fiandrino, C., Kliazovich, D., Bouvry, P., 2015. Energy-efficient computation offloading for wearable devices and smartphones in mobile cloud computing. In: 2015 IEEE Global Communications Conference. GLOBECOM, IEEE, pp. 1–6.

Reference, A., 2022. Int32.maxvalue field. https://docs.microsoft.com/en-us/dotnet/api/system.int32.maxvalue?view=net-5.0. Accessed: 2023-04-25.

Singh, L., 2022a. RMVRVM – a paradigm for creating energy efficient user applications connected to cloud through rest api. In: 15th Innovations in Software Engineering Conference. pp. 1–11.

Singh, L., 2022b. Source code for rmvrvm experiment mobile application. https://github.com/LavneetSingh/RMVRVM. Accessed: 2023-04-25.

Smith, J., 2009. Patterns - wpf apps with the model-view-viewmodel design pattern. https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern. Accessed: 2023-04-25.

Smith, S., 2022. Architect modern web applications with asp.net core and azure. https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/?WT.mc_id=dotnet-35129-website. Accessed: 2023-04-25.

Xia, F., Ding, F., Li, J., Kong, X., Yang, L.T., Ma, J., 2014. Phone2cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing. Inf. Syst. Front. 16, 95–111.

Lavneet Singh is a Ph.D. Scholar and Adjunct Faculty Member at Dhirubhai Ambani Institute of Information and Communication Technology (DA-IICT), Gandhinagar (India). Previously, he has worked in the IT industry (like Microsoft) for more than 20 years. His research interest include Software Engineering and Cloud Computing.

Saurabh Tiwari is an Associate Professor at Dhirubhai Ambani Institute of Information and Communication Technology (DA-IICT), Gandhinagar (India). Previously, he worked as a Postdoc Researcher at Mälardalen University (Sweden) and worked with Volvo Construction Equipments AB in the Model-Based System Engineering (MBT, MBD) area. His current research interests are Requirements Engineering, Empirical Software Engineering, Mining Software Repositories, Green Computing and HCI. He is also doing research in Software Engineering Education and Training to identify/devise innovative teaching methodologies to teach Software Engineering concepts.

Sanjay Srivastava received the M.S. degree in applied physics from IIT Delhi, New Delhi, and the Ph.D. degree in physics from University of California at Los Angeles. He is a Professor of Computer Science with Dhirubhai Ambani Institute of Information and Communication Technology, Gujarat, India. His research interests include network protocols, distributed computing, and subsystem designs for Internet of Things.