Refactoring towards Cloud-enabled Multiuser Mobile Apps

Arpit Christi
Oregon State University
chrsitia@eecs.oregonstate.edu

Michael Hilton
Oregon State University
hiltonm@eecs.oregonstate.edu

Danny Dig
Oregon State University
digd@eecs.oregonstate.edu

Michal Moskal
Microsoft Research
Michal.Moskal@microsoft.com

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Nikolai Tillmann
Microsoft Research
nikolait@microsoft.com

Abstract

Cloud computing has become a great tool for many developers to use, with many benefits, especially in mobile development. However, adding cloud functionality to an existing app can be difficult and time-consuming. We present CLOUDIFYER, a tool that can automatically refractor *local* data structures into *cloud* data structures on the TouchDevelop platform. As this platform targets students and hobbyist programmers, lowering the barrier of entry to using the cloud is imperative.

1. Introduction

Mobile apps have gained significant popularity in the recent years. According to Gartner [5], by 2016 more than 300 billion apps will be downloaded annually. The cloud has amplified the utility of mobile devices by providing additional computing power and storage, which translate into improved battery life [2], security [11], bandwidth utilization [14], location awareness [6]. In this paper we are focusing on another killer feature enabled by the cloud: the ability to develop rich, multiuser apps. Examples abound from domains such as social networking (e.g., Facebook, Twitter), multiplayer games, collaborative data collection (e.g., Citizen Science [4]).

To take advantage of the benefits of the cloud, app developers face a high entry barrier. They need expertise on many topics: communication protocols (e.g., web services, REST, SOAP, etc.), data storage (e.g., Amazon S3, Microsoft SkyDrive, etc.), databases, cloud infrastructure (e.g., Amazon EC2, Windows Azure, etc.), programming or scripting languages. Similarly, converting a single into a multiuser app has a high entry barrier: they need to determine the candidate data structures and methods that operate on data structures and move them to the cloud. Currently, this process is manual, time consuming, and error prone [7].

In this paper we are lowering the entry barrier to allow even hobbyists and beginner app developers to use the cloud. Thus, we are targetting TouchDevelop [13], a programming environment and language developed by Microsoft Research to write apps *on* mobile devices for mobile devices. We are employing automated refactoring techniques to convert local data structures into cloud data structures.

TouchDevelop introduced specialized cloud data types [1] that provide an abstraction layer over web service implementation, communication protocols, and storage. In order to make the app responsive, even when the connection to the server is unavailable, cloud data types provide both local copies of the data as well as eventually consistent sharable cloud storage. This paradigm allows programmers to use cloud types in a similar manner to local data structures, but to also enjoy the benefits of the cloud.

In this paper we present the results of our formative study to convert single to multi-user apps. We used three publicly available, single-user TouchDevelop apps (two productivity tools and one game) and manually converted them into multi-user apps. Thus we discovered four conversion steps: (i) identify local data structures that need to be shared between multiple users, (ii) for each identified local data structure add a new cloud data structure, (iii) replace local data structure API calls with cloud API calls, (iv) initialize the cloud data structures. Not all these steps can be automated; some (e.g., identifying data structures that need to be shared) require domain knowledge which is best provided by the app developer.

Using the lessons that we learned from the formative study, we designed and implemented a refactoring tool, CLOUDIFYER, to automate the conversion of local `Number Collection` into `Cloud Data Table`, and to transform the API calls. We selected this refactoring because its manual application is challenging: the developer needs to map a 1-dimensional, horizontal data layout (as in `Collection`) into a 2-dimensional data layout (as in `Table`). The difference is not only in the names of the APIs, but also in the cardinality of the mapping. Sometimes the mapping is 1-to-1 (e.g., `count` is the same in both data structure), other times the mapping is 1-to-many (e.g., `insert at` from `Collection` is transformed into a sequence of 3 operators from `Table`). In addition, sometimes there is no mapping, in which case it requires creating custom functions to achieve the same computation. For example, `Collection->max` needs to be converted into a custom function that iterates atomically over the elements of the `Table`. Such transformations can not be performed by a find-and-replace tool.

This paper makes the following contributions:

- **Idea:** To the best of our knowledge, we are the first to enable hobbyists and beginner programmers to tap into the power of the mobile cloud computing through the use of refactoring techniques.

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

- **Formative Study:** We have conducted a formative study on three real-world apps to learn what transformations are needed to convert single to multi-user mobile apps.
- **Tool:** We have designed and implemented the analysis and transformation algorithms to refactor *local* data structures into *cloud* data structures on the TouchDevelop platform.
- **Evaluation:** We have evaluated our tool, CLOUDIFYER, on a corpus of 109 mobile apps, resulting in 1139 transformations. The results show (i) that the refactoring is widely *applicable*: 94% of the candidate local `Collections` were successfully refactored into `Cloud Data Table`. Second, CLOUDIFYER saves human effort: on average it took 9 seconds for each performed refactoring. Third, CLOUDIFYER is *accurate*: 100% of the applied transformations are correct, and the tool correctly identified 95% of all necessary transformations.

2. Background on TouchDevelop Cloud Data Types

3. Formative Study

We selected three apps from the TOUCHDEVELOP script bazaar for our formative study. We converted them from single-user to multi-user apps. In this section we describe the script's behavior, the process to convert them, as well as the changes we needed to make.

The first app we selected was MILEAGE TRACKER **TODO:** replace all other usages of MileageTracker and other apps with macros, which is publicly available with the script id of **TODO:** change to citation `zenx`. Mileage Tracker is an app that records and calculates fuel usage in Miles per Gallon (MPG) and displays how it changes over time. We converted Milage Tracker into a multi-user app, such that multiple family members using the same family car can collect the fuel usage even across multiple devices.

Business Manager+ [**?**] is an app to track of business contacts. We converted it into a multi-user app so that business colleagues can share their contact list with others.

CliffHangers **TODO:** change to citation`jrqt` is a clone of the popular game "hangman." The player is presented with a series of blanks, and they must guess the letters that fill in the blanks to make a word. The player has a limited number of guesses, and if they cannot guess the word before they reach the limit, they loose the game. We converted it into a multi-player game so that two players can collaboratively work together on separate devices to guess the word.

After converting these apps, we ran them with multiple users to verify that in fact the apps work correctly.

Based on the lessons we learned from converting these three apps, we designed a process to convert a single to a multi-user app. Our process consists of four steps:

1. Identify data that needs to be shared between users
2. Create new Cloud Data Structures to hold the shared data
3. Replace the local usage with cloud usage for the shared data
4. Initialize Cloud Data

Next we will illustrate these steps using the MILEAGE TRACKER app. In step 1 we identified the data that needed to be shared: (i) the `MilageRecord` is a collection of numbers that holds the Miles Per Gallon for the past usage consumption, (ii) `MaximumRecordEntries` is a number that determines how many records should be stored, (iii) `UseUSUnits` is a boolean that stores the preference between metric or imperial units.

To illustrate step 2, let us consider one of the shared data structures, the `MilageRecord` collection. We created a new `Cloud Data Table` named `MileageRecordTable` to hold the data. Notice that the original data is stored in a one-dimensional data structure, whereas the `Cloud Data Table` is a two-dimensional data structure.

In step 3 we replaced all the uses of the local `MilageRecord` with uses of `MileageRecordTable`.

In step 4 we initialized `MileageRecordTable`. Since the data is now persistent on the cloud, we need to change the initialization code from eager to lazy in order to avoid erasing all the data every time the app is launched. In addition, we need to make a choice between using the cloud (i) as a backup for the data for one user across multiple devices("just me session") versus (ii) to enable collaboration between multiple users across multiple devices ("everyone session").

Notice that steps 1 and 4 require domain knowledge in order to choose and refine the proper end-user experience.

Based on the type of the input, local data that we needed to migrate to cloud, we have identified three kinds of refactorings. Each refactoring can be performed using the general steps that we identified above. First, we refactored primitive data types into the corresponding cloud-enabled primitive type (e.g., from `Number` to `Cloud Number`). Second, we refactored local `Data Table` into `Cloud Data Table`. Third, we refactored local `Collection` into `Cloud Data Table`.

When carrying out the refactorings, we noticed that steps 1 and 4 are hard to automate as they require understanding the original program and the desired end-user experience. Steps 2 and 3 have different degrees of complexity. The first two kinds of refactorings are trivial because there is a perfect match between the local data and the cloud data type, so the change is as simple as prepending the keyword `Cloud` to the variable declaration.

The third kind of refactoring is non-trivial: it requires changing the program from using a flat, one-dimensional `Collection` to a two-dimensional `Cloud Data Table`. The APIs are different enough, so that sometimes we needed to map one function call from `Collection` into a sequence of calls from `Table`, whereas other times we had to augment the API by writing new functions. For these reasons, we automated the refactoring.

Table 1 lists the total number of refactorings that we applied as part of our formative study.

Table 1. Total number of refactorings

4. Automated Refactoring

5. Evaluation

6. Related Work

7. Conclusion

8. Motivating Example

In this section we introduce a motivating example for refactoring data structures to the cloud. Consider a program called Mileage Tracker that allows the user to store their milage so that they can be aware of their fuel usage. The user can input how much gas was needed to fill his tank, as well as the miles driven since the last fill up. This allows the application to be able to calculate what was the miles per gallon since the last fill up. The script will store this information and allow the user to view their fuel usage over time. This application was written before the introduction of cloud data to the TouchDevelop platform, and so the way that the application was developed, the data was being stored locally in a local data structure. The developer did check the option which preserves the values of the data structure across multiple runs, so as long as user uses same device, he will have up to date information regarding its mileage usage. However, the script cannot work across multiple devices which does not support the ability for multiple family members using the same family car to share their information effectively.

8.1 Goals of Refactoring to Cloud

The developer of the MPG tracking application would like to take advantage of cloud functionality. Specifically, the developer would like to enable the users to be able to access their data on multiple devices and see the same data. The developer would also like to enable social, collaborative features such as allowing users to share their MPG values with other users of the app.

This program was written before the introduction of the cloud API's to the TouchDevelop language. The program does not use any cloud data types, and all data is stored locally.

Before the introduction of the cloud data types, the developer could select the "save between script runs" option for local data, which would persist the data structures across multiple runs of the script. This approach has several limitations which significantly reduce the usefulness of the application to the end user. One of the major limitations of this approach is that the information can not be synced between devices. The user would only be allowed to track their MPG's on one single device. This poses a problem if the user would like to upgrade their device,or if there phone was lost or broken. Also, this approach will not work if there are multiple drivers of the vehicle, and they would like to be able to collaborate on tracking their fuel efficiency. However, this would not be possible using the local data structure.

8.2 Problem

If the application had been developed using the TouchDevelop table data structure, converting to a cloud table is a trivial operation. It is as simple as selecting the table object and changing the type to cloud table. The refactoring becomes non-trivial if the user was using another data structure, such as a collection. A TouchDevelop collection is a data structure that consists of a collection of any type of object that TouchDevelop supports. This could be a primitive type such as Number or String, or it could be something as complex as a Camera object, or a Map object. The difficulty in refactoring an existing collection into a cloud data structure arises in the need to convert into a cloud table. The methods that collection provides and the methods that cloud table provides have some overlap, but are not consistent. In addition, there are methods for `Number Collection` that have no counterpart for cloud table, i.e. avg. Table 2 shows the operations that are available for `Number Collections`, as well as the corresponding operations for cloud table, if they exist. We have categorized the type of transformation that will be necessary to transform every possible collection operation into a cloud table operations. We will describe how we perform these transformations Section 10.

9. TouchDevelop

In this section we provide a brief overview of the TouchDevelop language, as well as some more discussion on how cloud data types are implemented.

9.0.1 Cloud Data Types

The cloud types implemented in TouchDevelop include both simple types, cloud integers, cloud strings, as well as the structured types of cloud arrays and cloud entities. The only way that these cloud types are exposed to developers is as cloud tables and cloud indexes. The cloud table can be used in a similar manner as the local table data structure. However, when a developer uses the cloud table, they also get an eventually consistent cloud version and the option to share the data across multiple users.

If the developer wished to take advantage of the cloud features for primitive data, they must encapsulate them into a table first. Table 3 lists the types of primitives that are allowed as column types of the cloud table.

9.0.2 sessions

There are two different types of cloud sessions that are offered by TouchDevelop. There is the default session, called the "just-me" session as well as the "everyone" session.

9.0.3 just-me session

The "just-me" session is meant to share data between all devices by the same user running the same script. Other users cannot connect to this session. When the MPG script is refactored to use the just-me session, the user will now have the opportunity to see all of their data regardless of which device they are on. This allows the user to have the application installed on their phone and tablet and to share the data across each.

9.0.4 everyone session

The "everyone" session creates a session that everyone running the script can connect to. All users of the script will see the same data. For the MPG example, the everyone session would allow users to see the MPG statistics of a community of users. All users would add their MPG data to the common table, and the statistics that they would see would be for the entire community. For example, when they see the average MPG, it would be the AVG for all the users of the script.

10. DataStructure to Cloud REFACTORING

This section presents the data structure to cloud refactoring. We explain CLOUDIFYER's workflow, as well as the three types of transformations that we have identified that are necessary for the refactoring.

10.1 Workflow

CLOUDIFYER is written in TouchDevelop, and can be run using the TouchDevelop platform. Once it is run, the user is presented with a picker to choose which script to refactor. This must be a script that is saved for that user, but any public script can be saved as your own script. When the user selects which script they wish to refactor, they are presented with all `Number Collections` that are a part of this script. The user can then choose which `Number Collection` they wish to refactor by typing in the name into a standard TouchDevelop wall prompt box. Once the user has typed in the name and clicked the "ok" button, the tool will automatically refactor all the calls to the selected `Number Collection`, and then present some output to the user, which includes how many of each type of transformation were performed, as well as the total time required to complete the refactoring.

10.2 Algorithm

We describe the algorithm that is used by the refactoring engine to refactor TouchDevelop scripts.

Step 1: Create a new Cloud Data structure that transformations will use and add it to the script.

Step 2: Traverse through the script and find candidate statements for refactoring.

Step 3: Refactor local data structure statements into corresponding cloud data structure statements.

To explain our algorithm, we will walkthrough the refactoring of a TouchDevelop script, BlockY World. BlockY World is a

Number Collection Operations	Cloud Table Operations	Transformation Type
`add`	`add row`	Indirect
`at`	`row at`	Indirect
`add many`	NONE	Function
`avg`	NONE	Function
`clear`	`clear`	Direct
`contains`	NONE	Function
`count`	`count`	Direct
`index of`	NONE	Function
`insert at`	`row at→value`	Indirect
`max`	NONE	Function
`min`	NONE	Function
`post to wall`	`post to wall`	direct
`random`	NONE	Function
`remove`	NONE	Function
`remove at`	`row at→deleteRow`	Indirect
`reverse`	NONE	Function
`set at`	`row at→valueName`	Indirect
`sort`	NONE	Function
`sum`	NONE	Function

Table 2. Table with the Number Collection API's and corresponding Cloud Table Operations

Supported Cloud Table Types
Boolean
Number
String
DateTime
Location
User

Table 3. Supported Types

game where our hero must find his way through a maze of blocks, while also being able to move blocks to change the environment around him. BlockY World uses multiple `Number Collections` named `stats`, `spestats` and `zones`. For our further discussion of algorithm, we assume that user has selected `spestats` as the `Number Collection` to refactor. In step 1, a cloud data type is created and injected by CLOUDIFYER into the BlockY World script. A new cloud data table is added for every refactoring to avoid conflicts if there are more then one refactoring per script. Then a column will be added to the table with the same name as the `Number Collection` that is being refactored. For `spestats`, the tool will produce a cloud data table with a single column whose data type is number and who's column name is `spestats`. In step 2, TouchDevelop's JSON AST tree representation is used to traverse script's AST. If any references to selected collection to refactor is found, the tool marks it as a candidate statement to refactor. In step 3, each of the candidate statement is then transformed into individual new statement that replace the call to the `Number Collection` with a the corresponding cloud data action.

10.3 Transformations

There are several categories of transformations that are needed to transform local data structures to cloud data structures.

10.3.1 Direct Transformations

These are the transformations where both the `Number Collection` and the cloud table have the same action. For example, both have an action "post to wall" which will, as expected, post to the scripts wall. This is the simplest of the transformations, as all that is needed to do is replace the object that the action is being called on. Figure 1 shows a line of TouchDevelop code that posts to the wall a `Number Collection` that is stored in the global variable named `MileageRecord`.

```
MileageRecord → post to wall
```

Figure 1. post to wall before transformation

```
Cloud Data table → post to wall
```

Figure 2. post to wall after transformation

Figure 2 shows the same line of code after it has been transformed by our algorithm. Now the cloud data table is being posted to the wall.

10.3.2 Indirect Transformations

The indirect transformations are slightly more complex then the direct transformations. The indirect transformation is when there are two actions that have similar behavior, but different names. For example, `add`, which adds to a `Number Collection`, and `add row` which adds to a cloud data table. Consider the case where the user of the previous example would like to add the current mileage to the list that stores all the past mileages. In order to add it to a `Number Collection`, the developer would have to use the `add` operator. This is shown in Figure 3.

```
MileageRecord → add(Mileage)
```

Figure 3. add action before transformation

```
Cloud Data table → add row → MileageRecord :=
Mileage
```

Figure 4. add action after transformation

Figure 4 shows the same line of code after it has been transformed by CLOUDIFYER. Now the actions are operating on a cloud data table. Since each element in the `Number Collection` is being mapped to a row in the cloud data table, before the number can be added a new row must be added to the table. TouchDevelop offers the ability to chain functions together, so we can chain the add and assignment operators. In the same line of code we can add a new row to the table, as well as set the value as the current Mileage that we wish to store.

10.3.3 Custom Function Transformations

The third type of transformation we have implemented is custom function transformations. These are needed when there is no action in Cloud Table that will achieve the same functionality as the action that exits in `Number Collection`. For example, the developer of the mileage tracker would like to calculate the average across all the sessions of the app. The `avg` action will return the average of all the numbers in the `Number Collection`. In order to transform a `Number Collections` that contain this action, we inject a custom function into the code that provides the same behavior as the `avg` action. Then we can transform a call from the TouchDevelop API to a call to our custom function. Figure 5 shows a line of source code where the `avg` action is being called, and then the returned value is being stored in a local variable `avgMielage`.

```
var avgMileage := MileageRecord → avg
```

Figure 5. avg action before transformation

Figure 7 shows the same line of code after CLOUDIFYER has performed the transformation. The action `Cloud Data avg` is a custom function that is generated by our refactoring engine that is needed to calculate the average of all the values in the cloud data table. This function also returns a value, so we are saving it in the same local variable. Figure 7 shows the helper function that is added to the script.

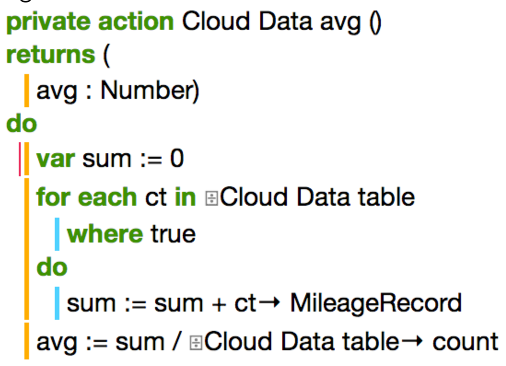
```
var avgMileage := Cloud Data avg
```

Figure 6. avg action after transformation

10.3.4 Unsupported Language Features

There were some transformations that we were unable to perform due to limitations of the TouchDevelop language. Local data structures, in our case the `Number Collection` data structure, can be passed as an input parameter (argument) or output parameter (return value) of a TouchDevelop action. Due to language constraints, cloud data tables must be declared as global variables and cannot be used as input or output parameters. CLOUDIFYER does not attempt to perform transformation in these situations. In the future, a transformation could potentially be developed that would transform the local structure from a parameter to a global variable, and then the local to cloud data transformation could be performed.

```
 private action Cloud Data avg()
returns(
avg: Number)
do
var sum := 0
for each ct in Cloud Data table
where true
do
sum := sum + ct → MileageRecord
avg := sum / Cloud Data table → count
```



```
private action Cloud Data avg ()
returns (
  avg : Number)
do
  var sum := 0
  for each ct in ⊞Cloud Data table
    where true
  do
    sum := sum + ct→ MileageRecord
  avg := sum / ⊞Cloud Data table→ count
```

Figure 7. custom function to provide avg functionality

`Number Collections` can be assigned to a variable using the assignment operator. However, cloud data tables are able to be assigned due to their global nature. We are therefore unable to transform any occurrences of a data collection that use the assignment operator.

CLOUDIFYER also does not support the custom function transformation of `add many`. We believe that this function could be implemented but it remains as future work. For this reason, CLOUDIFYER is unable to support transformations that involve the `add many` action.

11. Evaluation

To determine if CLOUDIFYER is useful we ask the following research questions.
Q1: **APPLICABILITY**: How applicable is the refactoring?
Q2: **EFFORT**: How much effort is saved by CLOUDIFYER when refactoring?
Q3: **ACCURACY**: How accurate is CLOUDIFYER when performing a refactoring ?

11.1 Methodology

To create a corpus, we wrote queries against a database of all publicly available scripts in the TOUCHDEVELOP script bazaar. We wanted to collect popular and mature scripts. Thus, our query returned top scripts (based on the number of times that each script was executed**TODO:** verify that this is what "number of runs" means) that contained a `Number Collection`. We retained the top 109 scripts.

We ran the refactoring in batch mode. From each script in our corpus, our tool randomly selects one `Number Collection` as the target for refactoring.

In order to determine the applicability of CLOUDIFYER, we report the number of scripts that our tool was able to refactor successfully, as well as the number of each kind of transformations performed inside a refactoring.

In order to determine the effort saved by using this refactoring tool we record the number of seconds each refactoring took.

After running the batch refactoring, we chose 20 script to manually evaluate to verify our results. In order to verify that our scripts did not introduce any runtime errors, we first ran the original script for three minutes and manually explored their functionality. We then ran the refactored version of the script. In order to determine the precision of our refactorings, we performed manual code inspections on these randomly selected scripts. We looked for all the potential transformations for each refactoring, and determined if CLOUDIFYER had correctly identified them. If they were correctly identified, we determined if the transformation had been applied correctly.

11.1.1 Further Validation

In order to further validate our results we randomly chose 20 scripts to further evaluate. We then ran the scripts for up to three minutes and explored their functionality. During this exploration we took notes of the behavior and any output of the script. We then performed our refactoring on each script and then ran each script for three minutes again to verify that our script had not introduced any runtime behavioral changes. We did not observe any runtime changes based on the changes to these scripts. In addition to running these scripts we also performed code inspections of these randomly selected scripts. We counted any potential transformations that were missed by CLOUDIFYER as well as we inspected all the transformations that had been performed to see if they had been performed precisely or imprecisely. Using these values we were able to calculate precision and recall for CLOUDIFYER. We define precision as

$$precision = \frac{|TruePositive|}{|TruePositive| + |FalsePositive|}$$

and recall as

$$recall = \frac{|TruePositive|}{|TruePositive| + |FalseNegative|}$$

A *TruePositive* is a potential transformation that is correctly identified as a transformation, and the transformation is correctly performed. A *FalsePositive* is when CLOUDIFYER applies a refactoring to a line of code that is not a potential target. In other words, it transforms code that should not have been transformed. *FalseNegative* is when a transformation that should be applied is not. This is when CLOUDIFYER does not correctly transform some code, leaving it as it was before, when in fact, it should have been transformed.

11.2 Results

APPLICABILITY: We determined that our refactorings are widely applicable to TOUCHDEVELOP scripts that contain the `Number Collection` data structure. Table 4 shows the results for running our refactoring tool on 109 scripts. Out of the 109 scripts that were attempted to be refactored by our tool, only 7 were unable to be refactored due to unsupported language features as described in Section 10.3.4. CLOUDIFYER was able to refactor 94% of the scripts that we attempted.

Total Scripts	109
Number Successfully Refactored	102
Unsupported Language Features	7

Table 4. Total Scripts Refactored

EFFORT: Figure 11.2 shows the number of transformations performed per script as well as the average times. Each refactoring consisted on an average of almost 12 transformations. The most common transformation was indirect transformations with 10 transformations on average for each script.

The last column shows the time in seconds. On average each refactoring took a total of 9 seconds to complete. The maximum length of time any single refactoring took was 18 seconds. As the number of transformations on average was 12, this shows that our refactoring tool saves significant effort, as the time to make these transformations by hand would clearly exceed a few seconds.

	Total Trans	Direct Trans	Indirect Trans	Custom Function	Time in secs
AVG	11.6	1.20	9.07	1.34	9.30
MAX	112	13	112	19	18

Table 5. Average Transformations

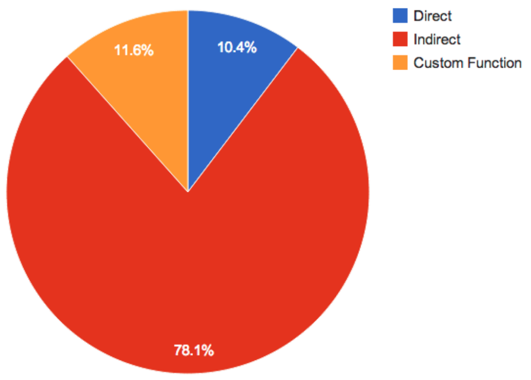


Figure 8. Transformations by Percentage

ACCURACY: By manually checking 20 scripts, we analyzed the precision and recall for our tool. We found a precision of 100%. We also calculated the recall, and our recall value was 94%. There were over 145 transformations that we inspected in these 20 scripts, and we only found 8 potential transformations that CLOUDIFYER missed.

Precision	100%
Recall	95%

Table 6. Precision and Recall values

11.3 Case Studies

11.3.1 Motivating Example

We forked the Mileage Tracker script and created our own Mileage Tracker script. Without making any logical change to the script, we used `Number Collection` instead of number map to store data. We then manually replaced a slice api call with corresponding logic as the slice api call is not available with `Number Collection`. We ran both the original script and our manual changes to ensure that both scripts exhibited the exact same functionality. The new Mileage Tracker script is available in TouchDevelop with script id `mfbb`. We then refactored the Mileage Tracker script using CLOUDIFYER. All the transformations were successfully applied and script did not have any compilation errors. Though to ensure that script behavior was exactly same as before we needed to make two lines of code change in InitializeApplicationSettings method of the script. As Microsoft TouchDevelop recent version removed is invalid with cloud data table, we have to replace it with corresponding logical step. The other one line of change in code is to enable everyone session on the cloud data table. This will allow multiple users with their own separate sessions to share same cloud data table. We ran the refactored Mileage Tracker application with three different devices, two laptops and one mobile phone to verify Mileage Tracker is indeed refactored into multi device, multi user application. Thus, with our automated refactorings and two lines of code changes we were successfully able to convert a single user, single device application into multi user, multi device application. The refactored Mileage Tracker Appliation is available in TouchDevelop with id `eyhca` and name Mileage Tracker Refactored.

11.3.2 BlockY World

To further demonstrate the usefulness of cloud refactoring, we applied it on BlockY World, a game script written for TouchDevelop. The BlockY World script id is skdk. 65 users are following this game and the game has been installed 350. To convert this game from single device to multi device/multi user game via refactoring, we used our tool to refactor all the collections to cloud data types. BlockY World has 8 `Number Collections` and one `picture collection`. As TouchDevelop cloud data tables do not support picture type, we are unable to convert the picture collection to cloud data type. All 8 `Number Collections` were automatically refactored into corresponding cloud data type using the tool. We also added some manual code to initialize and share the data.

However, we were unable to convert BlockY World into a multi device game. We throughly analyzed the script to see what additional steps would be needed to complete the transition to a multiplayer game.

Our further analysis of BlockyWorld suggests that some global variables also need to be moved to cloud because to make the script multi device some global variables also need to be shared across devices over the cloud. For instance, variables like old_x and old_y and $curr_x$ and $curr_y$ needs to be stored across the devices. These variables are of type number.

BlockY World consists of some variables whose cloud type representation is not available at this point of time. ercan is game board variable used for the BlockY World game. In order to effectively share the game board across multiple devices, we need to move variable ercan to cloud. Currently no cloud data type implementation supports game board variables. BlockyWorld consists many such variables whose cloud data type conversion is not possible.

We also determined that for this application, eventually consistency was not responsive enough for realtime gaming. Though the eventual consistency model works fine with many types of applications, it is not particularly suited for games where real time data needs to be shared quickly across device. TouchDevelop team is planning to support flush implementation for synchronization with stronger consistency model [1], where any changes in local data is immediately synchronized with cloud data, though it is not available currently. Though we are not able to refactor BlockY World into a multi device application through refactoring, analysis of BlockY World brought into light certain implementation level issues of both TouchDevelop as well as textsccloudifyer that hinders widespread practical usage of textsccloudifyer in case of realtime gaming application refactoring.

11.4 Threats to validity

Construct Validity: Does the evaluation of candidate scripts prove the applicability of our tool? Since the refactoring to the cloud is such a new part of the TouchDevelop language, there is not a lot of data for us to identify when developers will actually choose to refactor to the cloud. By showing that most of the scripts that use the local data structure are applicable candidates, we can show that there is wide applicability for our solution, should developers choose to use it or not.

Internal Validity: We minimized the threat of internal validity by performing a case study of apps that were written before the introduction of Cloud Data, and by using publicly available scripts, the developers were unaware of our study.

External Validity: Do our results generalize? We chose over 100 of the most commonly run scripts to use as our corpus, but we put no restrictions on the tags, or types of the scripts. Our selection include scripts tagged with entertainment,sports, games, education, productivity and platform bugs. Since tags are not required in TouchDevelop we observed many scripts without tags. However, in a addition to a wide variety of scripts tags, we also observed a variety of script types while performing the evaluations. We do not foresee any reasons why these results would not generalize to all TouchDevelop scripts.

Reliability: Is our evaluation reliable? The scripts that we used to evaluate our tool are all available on the TouchDevelop website, and CLOUDIFYER is published on the TouchDevelop website with a script Id of //TODO FINAL VERSION OF SCRIPT ID.

Very few scripts in TouchDevelop have any tests at all. We were not able to use these to verify that we did not break the runtime behavior of the scripts. In order to mitigate this we manually validated a random selection of scripts to ensure constant runtime behavior before and after our refactoring.

12. Related Work

Strauch et al. [12] provide a methodology for refactoring applications to move from application data to cloud data. Their approach is based on finding a migration scenario that depends upon requirements and then mapping the scenario to one of their existing cloud data patterns. Though this paper provided a methodology at a high level, the actual application refactoring is completely manual and left up to programmers. Also, moving application data to the cloud was also manual and programmer will have to be involved at multiple level to make this refactoring possible.

Ling et al. [9] provided a systematic refactoring approach using theoretic computing to formally define a mechanism to convert object oriented systems into service oriented architecture systems. The services then can be moved to the cloud to benefit from the elasticity of the cloud. Again, this approach leads to the use of manual refactoring and manual moving of services to cloud.

CLOUDIFYER is different from both these approaches in that it performs the refactoring in an automated manner, thus saving programmers from the effort of moving data structures manually to the cloud.

Kwon and Tilevich [8] proposed *Cloud Refactoring*, a tool to automatically refactor methods of an enterprise application system into services that can be then ported to the cloud. Their tool also determines if a method is a good candidate for refactoring via static analysis of code and dynamic traces generated using application execution. However, even though the methods are automatically refactored into services, they still must be moved to the cloud manually. CLOUDIFYER performs the refactoring automatically, but also handles the transfer to the cloud. We do believe that Kwon and Tilevich's approach is a big leap forward in right direction.

CloudCone [3], also attempts to harness the power of mobile cloud computing, but their approach is significantly different then ours in that they create clones on the cloud, and then move the execution of the application as well as it's state between the cloud and the device.

13. Conclusions and Future Work

In this paper we present CLOUDIFYER, a tool to refactor local data structures into cloud data structures. We have shown that it is able to automatically transform most usages of `Number Collection` into cloud data references. We are able to perform these transformations while still preserving the behavior of the application. Should the user choose to take further advantage of the cloud data features, in some cases only a few lines of code are needed to take advantage of the cloud data features to provide more functionality.

In the future, it would be nice to automate those changes, so in addition to being able to refactor their data to the cloud, we would like to provide an automated transformation tool that would enable multiuser features to be automatically added to scripts.

References

[1] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP 2012–Object-Oriented Programming*, pages 283–307. Springer, 2012.

[2] H.-Y. Chen, Y.-H. Lin, and C.-M. Cheng. Coca: Computation offload to clouds using aop. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 466–473, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4691-9. . URL `http://dx.doi.org/10.1109/CCGrid.2012.98`.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[4] J. P. Cohn. Citizen science: Can volunteers do real research? *BioScience*, 58(3):192–197, 2008.

[5] Gartner. Jan'14, `http://www.gartner.com/newsroom/id/2153215`.

[6] A. Kansal, S. Saponas, A. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 661–676. ACM, 2013.

[7] A. Khan, M. Othman, S. Madani, and S. Khan. A survey of mobile cloud computing application models. 2013.

[8] Y.-W. Kwon and E. Tilevich. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering*, pages 1–28, 2013.

[9] H. Ling, X. Zhou, and Y. Zheng. Refactoring from object-oriented systems to service-oriented systems: A categorical approach. In *Service Sciences (ICSS), 2010 International Conference on*, pages 214–218. IEEE, 2010.

[10] B. Manager. Jan'14, `https://www.touchdevelop.com/app/beta#list:installed-scripts:script:chgv:overview`.

[11] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *Proceedings of the First Workshop on Virtualization in Mobile Computing*, MobiVirt '08, pages 31–35, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-328-0. . URL `http://doi.acm.org/10.1145/1622103.1629656`.

[12] S. Strauch, V. Andrikopoulos, and T. Bachmann. Migrating application data to the cloud using cloud data. In *Proceedings of the 3rd International Conference on Cloud*, pages 36–46.

[13] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 49–60, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0941-7. . URL `http://doi.acm.org/10.1145/2048237.2048245`.

[14] C. Vemulapalli, S. K. Madria, and M. Linderman. Pre-distribution scheme for data sharing in mobile cloud computing. In *Proceedings of the First International Workshop on Mobile Cloud Computing &#38; Networking*, MobileCloud '13, pages 11–18, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2206-5. . URL `http://doi.acm.org/10.1145/2492348.2492353`.