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

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.

- **Formative Study:** We have conducted a formative study on 2 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*: 88% 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.

Cloud computing has been a long held dream of treating computing as a utility [**?**]. It can enable developers with innovative ideas for new Internet services to no longer require the large capital outlays in hardware to deploy their service, or human expense to operate it. By using a pay-as-you-go pricing model, resources are only paid for if they are needed, which lowers the up-front investment, and allows developers to avoid paying for unneeded resources [**?**]. Some examples of cloud computing servers include Amazon EC2, Windows Azure, and Google App Engine. While there are many benefits of cloud computing, there are also some disadvantages. As cloud services are often remote, they can suffer from latency and bandwidth related issues [**?**]. This provides a challenge to developers if they wish to provide a similar experience to users while still taking advantage of the benefits that cloud computing can offer. In addition, if a developer wishes to transform an existing application that previously had only dealt with data locally, there new must be a new communication layer that the developer must deal with in order to transform the application to use cloud services.

If a programmer decides to set up a cloud server, they will first have to determine the candidate data structures and candidate methods that operate on data structures to move to the cloud server. Then the programmer will have to program the desired functionality to run on the cloud server. Often this will involve using a completely different language and environment as the rest of the application. Some common technologies for communicating with the remote application are web services, REST services and WCF services. Building services using any of these technologies require the programmer to learn IDEs and programming languages that support these technologies.

Once the programmer is able to build a service to run on the cloud server, they still need to execute the manual steps of moving the services to cloud. Unless the programmer has access to a cloud server, they will most likely have to use a commercial service such as Amazon EC2 or Windows Azure. The programmer will also have to develop a way to persist the data on the cloud server. This would involve learning how to use cloud data storage such as Amazon S3 or Microsoft SkyDrive. Performing all these steps is clearly a challenge even for an experience user, as all these steps are manual, time consuming, error prone and may require multiple iterations of them before reaching a stable error free level.

TouchDevelop [**?**] is a programming environment and language developed by Microsoft Research which makes it possible to write applications on mobile devices. TouchDevelop was designed to engage users by allowing them to write scripts that show and manipulate items that connect with their personal interests and possessions, such as music and pictures on their phone, the sensors of the phone, and the friends in their social networks. TouchDe-

velop includes a typed, structured programming language that is built around the idea of only using a touchscreen as the input device to develop code. It has built in primitives that make it easy to access common sensors on mobile devices. While the language is similar to traditional text-based languages, the editor has enforced structure to help avoid syntax errors as users will be entering their data with the less than precise touch gestures.

Before the introduction of cloud types, TouchDevelop supported REST calls to web services. This allowed users to connect to an existing REST server and add functionality to their application. However, if the REST server did not already exist, setting up such a server would involve a significant amount of knowledge and effort. As TouchDevelop's target audience includes students and hobbyist programmers [?], this would almost certainly be more then could be reasonably expected from them.

In order to assist developers with introducing cloud functionality into their scripts, TouchDevelop introduced specialized cloud data types. [?] These cloud types were designed to provide an abstraction layer to free developers from concerns such as web service implementation, communication protocols, and local 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 constant sharable cloud storage. This implementation allows programmers to use cloud types in a similar manner to local data structures, but to also enjoy the benefits of the cloud.

To enable TouchDevelop programmers to take advantage of these new features, we designed, implemented and evaluated CLOUDIFYER, a novel tool that will perform refactorings from local data types to cloud data types in an automated manner. As far as we know, this is the first tool to perform then entire refactoring from local data to cloud data automatically. The rest of the paper is structured as follows. Section 2 presents a motivating example for this refactoring. Section 3 provides an overview of the TouchDevelop platform as well as some of the cloud data type implementation details. Section 4 describes the refactoring that we implemented. Section 5 presents the results of evaluating CLOUDIFYER by running it on 109 existing touch develop scripts, of which we were able to refactor 88%. For every refactoring we were able to perform an average of 11 transformations to each script in 9 seconds. We present a discussion of related work in Section 6. We finish the paper in Section 7 with our conclusions and future work.

2. 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.

2.1 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.

2.2 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 1 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 4.

3. 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.

3.0.1 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 2 lists the types of primitives that are allowed as column types of the cloud table.

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

Supported Cloud Table Types
Boolean
Number
String
DateTime
Location
User

Table 2. Supported Types

3.0.2 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.

3.0.3 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.

3.0.4 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.

4. 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.

4.1 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.

4.2 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 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.

4.3 Transformations

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

4.3.1 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.

4.3.2 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

Figure 4 shows the same line of code after it has been transformed by CLOUDIFYER. Now the actions are operating on a cloud

```
Cloud Data table → add row → MileageRecord :=
                    Mileage
```
Figure 4. add action after transformation

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.

4.3.3 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

4.3.4 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.

`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, CLOUD-

```
 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

IFYER is unable to support transformations that involve the `add many` action.

5. Evaluation

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

5.1 Methodology

5.1.1 Verification Corpus

We developed our refactoring tool as a TouchDevelop script. In order to empirically evaluate the application, we ran it on 109 scripts. In order for a script to be a candidate for refactoring it should have at least one NUMBER COLLECTION. In order to make our evaluation easier, we also required the NUMBER COLLECTION be a global variable. Our refactoring works on non-global variables, but using global variables simplifies the automated identification of refactoring targets. We chose to target the top ranked scripts based on number of runs.

For each script we ran a program that would identify all the NUMBER COLLECTION data structures in the target script. Once the NUMBER COLLECTIONs had been identified, one would then be chosen to be the target of the refactoring. For each successful refactoring we returned several metrics.

In order to determine the applicability of our tool we tracked the number of scripts that our tool was able to refactor, as well as the number of each specific class of refactoring.

In order to determine the effort saved by using this refactoring tool we record the number of seconds each refactoring took. Table 5.2 shows the refactoring times. The longest refactoring took 18 seconds, but the median time was only 9 seconds. The majority of the time spent doing the refactoring was spent locating transforma-

tions, so the time that the refactoring took was correlated with the program size, not with the number of transformations performed for each script.

We did find some programs that our tool was unable to refactor. Currently the way Cloud Data tables are implemented in TouchDevelop, they cannot be passed as arguments to a method, they may only be used as a global variable. Since NUMBER COLLECTIONs do have the ability to be passed as collections, the tool is unable to refactor programs where the NUMBER COLLECTION is passed as an argument. We only found 5 scripts that were unable to be refactored for this reason, as noted in Table 3. We also were unable to implement the `add many` functionality, so any script that called the add many action on a NUMBER COLLECTION was unable to be transformed by our tool.

In order to determine if the tool was accurate or not we chose 10 applications to manually evaluate from the list of scripts we had run our automated refactoring tool on. In order to evaluate the accuracy we first checked to ensure that our automated tool had not introduced any new syntax errors into the script. After that, we looked at each reference to the original NUMBER COLLECTION and ensured that it had been correctly transformed into the correct call to the Cloud Table.

5.1.2 Further Validation

In order to further validate our results we randomly chose 10 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.

5.2 Results

APPLICABILITY: Table 3 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 4.3.4. CLOUDIFYER was able to refactor 91% of the scripts that we attempted. This shows that CLOUDIFYER is clearly applicable to a wide range of scripts that contain the NUMBER COLLECTION data structure.

Total Scripts	109
Number Successfully Refactored	98
Unsupported Language Features	7
Tool Crashed on Refactoring Attempt	4

Table 3. Total Scripts Refactored

EFFORT: Figure 5.2 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

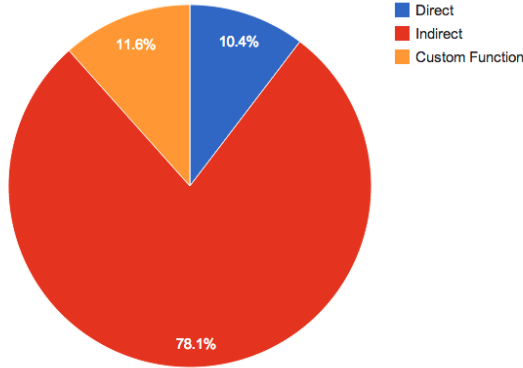**Table 4.** Average Transformations

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 5. Precision and Recall values

5.3 Case Studies

5.3.1 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.

5.3.2 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 multi-player 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 [**?**], 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.

5.4 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.

6. Related Work

Strauch et al. [**?**] provide a methodology for refactoring applications to move from application data to cloud data based on finding a migration scenario depending upon requirements and then mapping the scenario with one of the 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 cloud was also manual (actual move of data to cloud) and programmer will have to get involved at multiple level to make this refactoring possible.

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

Different from these two approaches, our tool achieves refactoring completely automatic and programmers are saved from the effort of moving data structures manually to cloud. Once a data structure is selected to refactor, the outcome of our tools is cloud data structures and updated code that references the cloud data structures.

Kwon and Tilevich [**?**] proposed Cloud Refactoring, a tool to automatically refactor methods of an enterprise application systems into services that can be then ported to cloud. Their tool is even capable of determining if a method is a good candidate for refac-

toring depending upon static analysis of code and dynamic traces generated using application execution. Still, once methods are automatically refactored into service, manual movement of services to cloud is needed, something that our tool achieves automatically. We still believe that Kwon and Tilevich's approach is a big leap forward in right direction.

The approach of partially moving computations of mobile device to a clone virtual device is a very recent addition. One such approach CloudCone is discussed by Chun et al. [**?**], but it significantly differ from our approach as it ports part of execution to a cloned device in cloud at run time.

7. 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.