Beyond Open Source: The TouchDevelop Cloud-based Integrated Development and Runtime Environment

Thomas Ball Sebastian Burckhardt Jonathan de Halleux Michał Moskal Nikolai Tillmann
Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{tball,sburckha,jhalleux,micmo,nikolait}@microsoft.com

ABSTRACT

The creation and maintenance of mobile applications is a complex process, involving a set of technologies ranging from an application programming language and integrated development environment (IDE), through the deployment and update models of the various app stores, to a cloud back end that stores user and telemetry data (with its own programming language and a variety of hosting issues).

We present the design of a Cloud-based Integrated Development and Runtime Environment (CIDRE) to make the creation of mobile+cloud applications easier for non-expert programmers. A CIDRE combines an online programmer community, a browser-based IDE, and an app store. The deep integration of the three elements decreases friction in the software engineering of apps through automated source control and dependency management, an open architecture for distributed plugins, and a crowd-sourced collection of crash reports and coverage/profile data. A CIDRE brings together three audiences: language/IDE designers (the authors of this paper), application programmers, and application users, who can easily modify the apps they are using (all distributed in source), and become programmers themselves.

We have implemented a CIDRE in the form of TouchDevelop, a streamlined, cross-platform, browser-based programming environment. We describe the design of TouchDevelop and the use of automation at various points to make it easy for the three parties to communicate and give feedback to one another. As we will show through our analysis of three years of deployment of TouchDevelop to hundreds of thousands of users, much of the design of our CIDRE can inform other approaches to cloud-based software engineering.

Categories and Subject Descriptors

D.2.6 [**Programming Environments/Construction Tools**]: Integrated environments; D.2.3 [**Software Engineering**]: Program editors

General Terms

Design, Languages

Keywords

IDE, cloud, plugins, ASTs

MSR-TR-2014-63

1. INTRODUCTION

Application (app) stores for mobile platforms such as tablets and smartphones have lately become very popular. Even though the majority of apps available in these stores are conceptually very simple, there can be a lot of friction in the development, deployment, and maintenance of these apps. The hopeful app creator must first master a complex set of technologies, including general purpose programming languages (be it Java, C#, or even a memory unsafe language like Objective-C) and integrated development environments (IDEs). The next step is understanding the deployment and update models of the various app stores. Finally, many mobile apps require a cloud back end to store user data and telemetry data not provided by the app store, which requires gaining mastery of cloud infrastructure, and possibly another language for server-side programming. For many people, this puts mobile app creation out of reach.

To address this complexity, we propose to merge the concepts of an app store, an IDE, and a programmer community into one deeply integrated environment, dubbed a Cloud-based Integrated Development and Runtime Environment (CIDRE). A CIDRE supports the five key attributes of an *emerging experience* (a more precise description will follow in § 2.1):

- **Cross Platform and Mobile**: the IDE and runtime experience are available across platforms and on various form factors, including smartphone, tablet, laptop, and desktop;
- **Online Community**: programming is a social activity, as evidenced by the popularity of sites such as `stackoverflow.com` and `github.com`; a CIDRE integrates social features (reviews, scores, forums, comments, ratings, collaboration, crowdsourcing) to foster collaboration, encourage growth of the programmer community, and ease the entry for newcomers;
- **Ubiquitous Workspace**: users expect their workspaces and documents to move with them as they change devices, so a CIDRE must replicate a user's programming workspace across a user's devices, which leads to the topic of...

- **Offline Support**: a CIDRE is not simply a shell into services in the cloud; it provides substantial offline functionality so that a user may continue to work productively even when disconnected from the cloud;
- **Secured Identity**: supporting online interactions and a ubiquitous workspace requires that users assume an *identity* that can persist within the system.

We argue that a CIDRE's support for an emerging experience brings many automation and simplification opportunities, greatly reducing the friction for mobile+cloud app development.

We have implemented a CIDRE in the TouchDevelop project (`touchdevelop.com`) [14], which originally focused on bringing general-purpose programming capabilities to anyone with a smartphone, without the need for a separate computer or a physical keyboard. The driving idea was that smartphones often will be the first, and possibly the only, computing device people have access to, and that these modern equivalents of 1980s 8-bit computers need a modern equivalent of BASIC, complete with access to the on-board sensors, graphics, and the web.

Over the past three years, the project expanded from a single smartphone app into a web app running in all modern HTML5 browsers on phones, tablets, and keyboard-equipped PCs (regardless of the operating system they may be running). Additionally, TouchDevelop gained capabilities normally found in general purpose languages and IDEs, while maintaining the simplicity of the initial simple smartphone app: an open source publishing system with automated updates and dependency tracking, a vast library of built-in functions, an advanced module system for user-defined components, user-defined types with automatic cloud replication, a literate programming system, a single-step and breakpoint-based debugger, a profiler, a crash-logging system with bucketization, and crowd-sourced coverage collection.

The TouchDevelop CIDRE brings together three audiences, as shown in Figure 1: the CIDRE designers (the authors of this paper), app programmers, and app users:

- **CIDRE Designers**: The TouchDevelop IDE is delivered as web app to TouchDevelop app programmers. We use the cloud to deliver multiple versions of the IDE simultaneously; TouchDevelop programmers who have a high experience score have the option to use the *beta* version, which includes new features not in the *current* version. The IDE has a high level of instrumentation and telemetry that sends data back to the cloud to help us understand the stability of a new release and the usage of features. Assertion failures in the IDE automatically generate bug reports, allowing us to quickly see if a new feature is causing problems.
- **App Programmers** can rapidly build apps and iterate with their users, as well as with the CIDRE designers to get bugs fixed and/or questions answered; Furthermore, just as the TouchDevelop IDE contains instrumentation, the TouchDevelop compiler inserts instrumentation into apps so that programmers automatically get profiles and coverage information as users execute their apps. Runtime failures in a TouchDevelop app present users with a dialog whereby they can submit a stack trace to the app programmer.

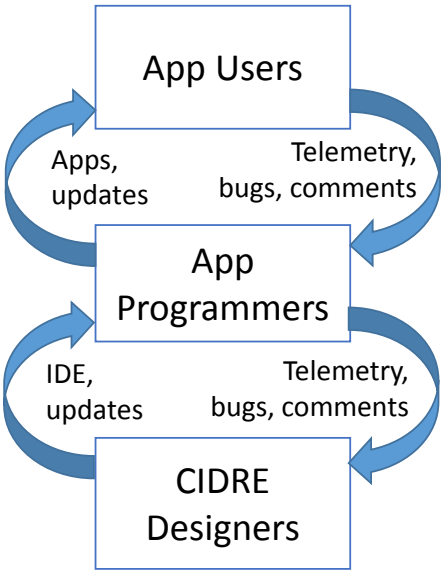

Figure 1: CIDRE creates positive feedback cycles between designers, programmers and users.

- **App Users** can provide feedback to programmers in comments and bug reports, as mentioned above. But, more importantly, app users are empowered to become app programmers as the TouchDevelop IDE is just one tap away from the app store, allowing any app to be modified and re-published in a matter of seconds. TouchDevelop tracks the provenance of scripts, so that a user who authors a script that is changed and re-published by others can get an increased ranking.

The evaluation of the data we have collected, including over 100,000 published scripts and associated information publicly available via REST APIs (see § 3.7) shows the CIDRE approach pioneered in TouchDevelop to be largely successful: Users are creating apps for mobile devices, they are modifying the apps they use, and taking advantage of libraries published by other users. Most importantly, the quality of apps created on mobile devices in on par or better than the ones created on desktop devices. Moreover, users who use both mobile and desktop devices are by far the most successful at app creation.

Outline.

In the reminder of this paper we first make the idea of CIDRE more precise (§ 2), then describe an implementation in the form of TouchDevelop (§ 3), and finally present an evaluation of three years of data collected about our user community and their apps (§ 4). We finish with discussion of related work (§ 5) and conclusions (§ 6).

2. CIDRE

We now clarify what we mean by a cloud-based integrated development and runtime environment (CIDRE). A CIDRE has the following three key characteristics:

1. it integrates the development, storage, and the deployment of applications — thus combining an IDE, code repository, and app store.

2. it provides all components of the *emerging experience*, as defined in §2.1 below.

3. it supports the development of applications that provide the emerging experience to their users.

In the remainder of this section, we define the emerging experience, which is independent of the chosen particular application domain. We then discuss how the CIDRE's deep integration of features provide these emerging experiences, both for the CIDRE itself, and for the applications being developed.

2.1 The Emerging Experience

The explosive growth of mobile devices and cloud-based services has changed the way that users experience applications. As technologies advance, the distinction between desktop applications, mobile apps, and websites is becoming increasingly blurred — applications can use cloud services to look more like websites, websites can employ HTML5 to look more like apps, and apps can provide rich functionality on PCs to look more like applications. What results from this confluence of applications, mobile apps, and websites, is a new combination of experiences, which we call the "emerging experience", defined as the combination of the following five components:

Cross Platform and Mobile. While most users still rely on a PC or laptop when it comes to "serious work", there is an increasing expectation that applications make some or all of their functionality available for use on mobile devices, such as tablets or phones. We expect this trend to continue. Pragmatically, the question we should ask is not whether users prefer PCs or mobile devices, but how we can provide a good experience on both. This can be quite challenging because (1) there is an extreme variation in screen sizes among phones, tablets, and multi-monitor PCs, and (2) tablets and phones are intended to be used without a keyboard, for the most part. The input paradigms are shifting towards touch and speech. In the future, they may shift further towards various natural interfaces based on cameras and sensors, or even direct brain interfaces.

Online Community. When chat rooms, e-mail, and instant messaging started, they were isolated services and not tied to particular applications. However, since using a computer in general appears to be more productive and fun in the context of an online community rather than just as a solitary enterprise, social features such as reviewing, forums, comments, ratings, collaboration, crowdsourcing, and gamification are now increasingly becoming integrated with all sorts of applications, apps, and websites.

Ubiquitous Workspace. Users no longer rely on a single device for running applications. Often, they have multiple PCs and/or laptops (at work and/or at home), they carry a smartphone at all times, and they use a tablet on the go or in recreational spaces. Thus, they are likely to frequently switch between devices, or even use multiple devices at the same time. As they do, they expect continuity. In particular, they expect that their data and preferences (the personal *workspace*) are synchronized between devices, and backed up in the cloud so that they remain continuously available even if some or all of the devices are powered off.

Secured Identity. Supporting online interactions and a ubiquitous workspace requires that users assume an *identity* that can persist within the system. This identity may or may not be revealing the true identity of the user. The identity must be secured by appropriate means, to ensure authenticity of online interactions and the privacy of the personal workspace. The identity is also required to support billing, which can be any combination of a one-time payment, recurring service fees, or micro-transactions.

Offline Support. Mobile devices often encounter an unreliable and/or slow internet connection, and are sometimes even completely disconnected. Thus, it is important that critical functions of the application remain available, and that unavailable functionality is exposed gracefully (because users have little patience for crashing or unresponsive applications).

Being serious about supporting the emerging experience in a CIDRE has far-reaching consequences. Mainstream programming with traditional IDEs presents some serious obstacles to achieving these goals: for example, the traditional editing experience for text-based code is ill suited for mobile devices with small screens and no keyboards.

2.2 Integration Benefits

We now discuss some of the benefits that result from a deep integration of code development, publishing, and social networking.

Open Collaboration. Since a CIDRE has complete knowledge of all the code and libraries needed to run an application, it is very easy for anybody to fork a published application to make modifications. This enables spontaneous collaboration and encourages users to openly exchange code (either in the form of published applications, or as libraries).

Telemetry and Feedback. A CIDRE integrates not only the development and testing of an application, but the whole lifecycle, which also includes publishing of the application in the app store, and the publishing of updates. Thus, a CIDRE can automatically collect telemetry information and report it to the developer. For instance, it can report how many users have run a script, it can display detailed code coverage, and it can report crashes. It also collects user ratings, comments and suggestions.

Onboarding and Crowdsourcing. Since a CIDRE includes an online community, it eases the process of learning about a new language and development environment. Over time, this effect accumulates as power users contribute documentation and tutorials, and answer questions in forums.

Powerful Runtime Services. To support users in building applications for the emerging experience (§2.1), we can bundle the services already used by the CIDRE into the ap-

plication runtime. For example, we can provide high-level APIs to support (1) working with user identities, (2) defining personal workspaces that synchronize automatically between devices, and (3) directly provide access to common cloud services, such as search and maps. For all of these, the integration is valuable because it provides simplicity and continuity (for example, the user identities at runtime match the identities in the CIDRE, and users need not re-enter personal information and passwords separately for each application) and can help to solve tricky data management issues, such as automatic schema migration for personal workspaces or cloud sessions when pushing application updates.

Research on Software Engineering. Last but not least, the use of an integrated environment simplifies the collection of data across the whole lifecycle of an application, and allows us to quickly incorporate lessons learnt. In TouchDevelop, we make all versions of all applications and variations (by all authors) publicly available for research, accessible via the web (see § 3.7).

3. TOUCHDEVELOP

This section describes TouchDevelop [14], an implementation of a CIDRE. TouchDevelop consists of two parts—the cloud back-end implemented primarily in C# on top of Windows Azure storage and compute services, and the client IDE, written in TypeScript (a typed extension of JavaScript [3]), that runs in all modern web browsers.

On Windows Phone and Android, TouchDevelop also is available as a native app in the respective app stores. The app makes certain platform (*eg.*, media libraries and sensors) available to the embedded web browser that runs the TouchDevelop client, which is otherwise identical to the plain web app (see [11] for detailed discussion of the approach taken).

We now describe the fundamental design decisions relating to code representation, the programming language, and how we use a general notion of publication to store a wide variety of entities in the cloud, including script versions, art resources, comments, reviews, tutorials, and documentation.

3.1 Code Representation

Relying on *text-based programming* in a CIDRE is problematic. Main-stream programming is still firmly rooted in ASCII representations, with minimal embrace of Unicode. However, editing program text using touch-based keyboards on small screens is vastly impractical; the use of special symbols, indentation, punctuation, long identifiers, and the required editing precision, make for a painful experience. We believe this to be the primary reason why touch-based devices have so far seen little use for editing code.

Our solution is to make the editing a bit more organized, using a semi-structured code editor: statements are manipulated directly at the level of the abstract syntax tree (AST)[13], whereas expressions are edited as a sequence of tokens. Semi-structured editing prevents syntax errors that span several lines of code. It also allows for editing expressions with an adaptive on-screen keyboard at the token level, avoiding the restrictiveness of a fully structured editor (in the vein of Scratch [10]). For expression editing, buttons represent tokens, not characters, and can change dynamically based on cursor context. Experiments by colleagues in the PocketCode project show that the mix of tree-based and token-based editing is more effective than solely tree-based editing [8].

Since editing is not based on a text representation, we can store and edit programs directly as abstract syntax trees, without superfluous formatting or punctuation. This opens several opportunities for improving the experience further.

First, appropriate interfaces can be designed to edit the code using various input modalities. Whereas touchscreen users get a context-sensitive, token-based on-screen keyboard as described above, keyboard users get an analog of traditional IDE auto-completion—when they type, a search is performed on the choices that would be proposed by the on-screen keyboard. Additionally, most common text editing operations (copy, cut, paste, indent a block, etc.) have tree-level equivalents and can be mapped directly. The multi-input-modality works hand in hand with workspace synchronization. For example, the user can develop a program on a big desktop computer, and then tweak it on a mobile device.

Second, because the user is editing the AST, the edit operations can be tracked much more precisely. For example, identity of statements can be preserved when they are moved, allowing for a more precise diff and better merge semantics: merges based on AST modifications rather than text edits can guess intentions better, and fail less often [4]. This is important for simplifying collaboration.

Third, code can be rendered automatically, using colors, indentation, and fonts consistently across all projects. Also, code can be rendered more space-efficiently, which helps to preserve precious screen space on mobile devices.

Finally, IDE plugins can easily manipulate ASTs directly, simplifying both code analysis and modification. This also lowers the entry barrier for research on software engineering and on refactorings.

3.2 Programming Language

The simplification of the IDE alone is not sufficient to enable the convenient development of applications on a mobile device. Because screen space is limited, it is important to keep code as *simple and concise* as possible. Clearly, low-level languages like C with explicit memory management and buffer overruns would be a bad choice. Also, using a mix of languages and formats (JavaScript, HTML, CSS, SQL, XML, XSLT) would require a proliferation of editors and editing modes.

TouchDevelop uses a custom designed, statically-typed programming language. This is dictated by the needs of the semi-structured code editor as well as the desire to keep programs concise, simple and high-level, yet flexible for research purposes.

Strong Typing. Because the language is statically typed, we can use the type of an expression to determine what the user may want to enter next. Thus, we can display type- and context-dependent buttons for editing, which is crucial to avoid typing characters.

Cloud State. Scripts need cloud storage to provide workspaces and online interactions for their users. However, writing and managing cloud services for this purpose can be quite daunting. In TouchDevelop, we have made this process easy, by using the cloud types programming model [6]. Users simply mark the data that is to be shared in the cloud (in a personal workspace, or with other users). This data is automatically replicated across devices and can be read and written efficiently (i.e. without applicaiton-level synchro-

nization or communication, online or offline). Our runtime ensures that all replicas are automatically synchronized and conflicts are resolved automatically and consistently, ensuring causal eventual consistency [7].

APIs. We provide numerous platform- and browser-independent APIs for accessing sensors (such as location, microphone, camera, gyroscope), media libraries (music, pictures), and web services. This ensures that conceptually simple scripts (e.g. pick a random song and play it, or take a picture and post it online) are indeed simple to write. TouchDevelop provides over 1,500 functions in its API set.

3.3 Storage and Publications

Apps in TouchDevelop consist of immutable modules which we call *scripts*. Scripts are the most important among various types of immutable *publications* that TouchDevelop users can create. Every publication is assigned a *unique id* upon creation. The following publication types are straightforward analogs of concepts from a programmer community or an app store.

- **scripts**: Scripts can be easily made public and shared with other users. The identity of the current author and the ids of script from which the current script was derived (*base script*) if any, is included in the script at the moment of publication. Scripts can be marked a **libraries**, and then referenced from other scripts.
- **comments** are free-form pieces of text attached to other publications. Comments attached to another comment form a thread of discussion. Comments can be also tracked as bugs or feature requests.
- **reviews** are much like star-ratings in an app store—they provide a quantifiable measure or quality of a publication (*eg.*, a script, a comment, or an art resource). In TouchDevelop a user can express that they like a publication by giving it (up to one) heart. We do not support negative reviews.
- **screenshots** are attached to scripts and can be posted by the author or other users.
- **art resources**: Apps usually use images and sounds in addition to source code. Much like scripts, these can published as immutable entities, and have comments and reviews attached later.
- **forums**: There are a number of forums where users can provide comments. As elsewhere, nested comments can be attached forming discussion threads.

Variations are scripts derived from another script (called the base script). The **base chain** of a script consists of the script itself and the base chain of its base script if any.

Every authorized **user** of the system has a corresponding id and an associated user publication, where the publication is mutable—the name, profile picture, etc. can be all updated.

The unification of IDE with programmer community and app store allows additional interesting publication types:

- **crash reports**: When a script crashes, the users are given an option to inform the author by posting a crash report (including free-form text if needed). These reports are then automatically bucketized by stack trace.
- **profile and coverage information**: Some script runs are randomly selected for profiling or coverage collection. All such information is automatically aggregated and attached to the script for everyone to see. Information about the total number of runs and installations also is included, forming another quantifiable measure of script quality.
- **documentation scripts** use literate programming to render scripts as documentation topics. This way documentation can be easily user-sourced. Documentation scripts can be made into interactive tutorials, see § 3.5.
- **plugin scripts** let TouchDevelop programmers extend the IDE. Plugins can be invoked from various points in the IDE and operate on the AST of the current script. They run locally in the IDE, but can invoke web services to do the actual work, giving rise to distributed **cloud plugins** (see § 3.4).
- **remixes** are scripts with more than one author in their base chain. A *direct remix* is a script authored by a different user than its base script.

The immutability of scripts plays a critical role in **dependency management**. Just before a script is published, all its dependencies (scripts used as libraries and art resources) are automatically published as well (if they have been modified), and their resulting identifiers are stored in the script. This way, every script has a consistent snapshot of its dependencies.

While scripts are immutable, there is a way for an author to publish an **update** to their own script. The users of the script (be it a top-level app or a library) are then notified and given an easy way to update.

Scripts are installed into user's **workspace**. Once installed, they can be modified, and ultimately published. The state of the workspace is **automatically synchronized** between different devices of the same user.

Clearly, many features of TouchDevelop (in particular, interactions with the online community, and the browsing and search of published scripts, art, documentation, and forum) depend on a connection to the cloud. However, the entire IDE functionality (including code editing with temporary storage, compilation, running, debugging, and profiling) is fully available offline because it is implemented completely on the client side.

3.4 Cloud plugin architecture

A very interesting, if not yet fully explored, area of research in CIDRE is the idea of distributed cloud plugins. Plugins are authored as TouchDevelop scripts, avoiding security issues with foreign JavaScript code running under the domain of TouchDevelop and dramatically lowering the barrier to entry for plugin authors. For example, a script to rename top-level declarations while updating the references (*eg.*, to enforce a naming convention) is six statements long. Additionally, plugins being scripts can be forked, reviewed, and commented on as usual.

Technically, the AST maps to a JSON object, which can be inspected and modified using regular JSON APIs. This enables processing of the data on a different server, not affiliated with TouchDevelop. The plugin script just needs to take the JSON object representing the AST, send it to the

server, and possibly save the updated AST back. Such cloud plugins can alleviate problems with limited computational capabilities of the phone, for example to run various static analysis of the code. We expect plugins to be particularly interesting for research purposes: TouchDevelop provides an easy way to deploy such plugins, an eager user-base (which could use lots of help with their programs), and any data about the plugin usage can be easily captured. Moreover, the programming language is small and the scripts are not too complicated, making it easy to develop prototypes.

3.5 Automatic tutorials

Automatic tutorials offer a gentle introduction to the IDE for user who are new to TouchDevelop and/or programming in general. The tutorials are authored as documentation scripts. A tutorial script is broken into steps, and for each step the IDE extracts the description and the target script. It then first displays the description and then guides the user through the process of creating the target script. Once the target script is reached, the IDE moves to the next step. This automated guidance works by continuously computing the diff between the ASTs of the target script and the current script, finding the first token that needs to be inserted or removed and then displaying tips on the UI elements that need to be tapped to accomplish this (this may involve navigating between different screens in the IDE etc.). For each completed step in a tutorial, the user receives positive feedback from the IDE.

Tutorials are fully automatic and adaptive—if the user decides to do something else, the tutorial engine will let them explore and then direct them back on track. The AST diff need not be exact—for example, it can ignore the exact values of string literals and the exact art resources used.

TouchDevelop, with the automatic tutorials, was listed as one of the options for the Hour of Code event (see `csedweek.org/learn`). During a week in November 2013 over 130,000 students took the tutorials.

3.6 Gamification

We have experimented with various schemes of bringing gaming components into the programming process. Scripts are ranked according to the number of hearts they receive, and the number of times they are installed and run. This score decays over time to let newer scripts rise to the top.

Additionally, users are assigned score based on the number of hearts their publications receive, the number of followers they have, the number of different features of TouchDevelop they have used, and the number of days they were active. The score is displayed prominently on the user profile, and we have anecdotal evidence of users taking it very seriously.

3.7 Openness

All the publications (scripts, comments, art resources, user meta-data, crash, coverage, and profile reports) are publicly available via the cloud REST APIs described at:

`https://www.touchdevelop.com/help/cloudservices`

This data has been used in several external research projects so far.

Scripts are in public domain from the point of publication on and can be freely forked by any TouchDevelop user. The plugin architecture lets anyone extend the platform, and we expect to introduce more plugin hooks in the future. Additionally, users contribute to documentation (including the automatic tutorials), report bugs and request features in the IDE, and provide answers on the forums.

On a few occasions the ability to fork scripts caused friction with our users. Overall, however, we found the open and public architecture to be conducive to development of a vibrant user community.

TouchDevelop also provides an export feature targeting various native app stores, including Windows, Windows Phone and Android, with more coming in future. So far about 1000 TouchDevelop-generated apps have been published in the Windows stores. Interestingly, one of the most popular educational games in the Windows Store, MindSticks (`mindsticks.com`), was authored in TouchDevelop.

Timeline.

The first release of TouchDevelop was in April 2011 as a native C# Windows Phone app, six months after the project started, and did not include much in the way of cloud support. Script publishing was added in August 2011, comments and reviews were added in November 2011, and libraries in February 2012. Around that time the rewrite of the IDE as a TypeScript web app started, which was finally released in October 2012. Debugger, crash logging, and coverage collection were added in July 2013 and profile collection a short while later. Cloud data was released in October 2013, and interactive tutorials in November 2013.

4. EVALUATION

This section presents an evaluation of the TouchDevelop platform and the two feedback cycles from Figure 1. § 4.1 discusses the IDE itself and interaction of the authors of this paper with app programmers and users, while the remaining subsections focus on app programmers, their creations, and interactions with app users.

4.1 TouchDevelop itself

As mentioned previosuly, TouchDevelop consists of a cloud back-end consisting of about 100KLOC of C# running in Windows Azure, and a client running in HTML5 web-browsers. The IDE part of the client (AST operations, compiler, editor, debugger etc.) is implemented in about 60KLOC of TypeScript; the runtime libraries are 50KLOC.

Every source code check-in to the client is built automatically and uploaded to the cloud back-end generating a uniquely named release. At every time there is one release labeled “current” and (a possibly different) one named “beta”. The “current” release is moved about every week, and the “beta” is moved several times per week. Users with high score are encouraged to try the beta version.

In case of an unexpected exception or assertion failure in the client, we log a crash report in the cloud. The client also sends instrumentation telemetry data. As of April 2014, we collect about 100 crash reports and 35,000 telemetry reports per day, from about 1000 users. These numbers have been growing steadily over time.

We found the crash reports to be tremendously useful during development. The cloud back-end automatically bucketizes crash reports by stack trace and type, and sorts them by number of occurrences. This lets us focus on the crashes occurring frequently and impacting more users. Crash reports include the current script being edited and 1000 or so recent log messages and instrumentation events. The log is particularly useful in presence of async APIs in JavaScript,

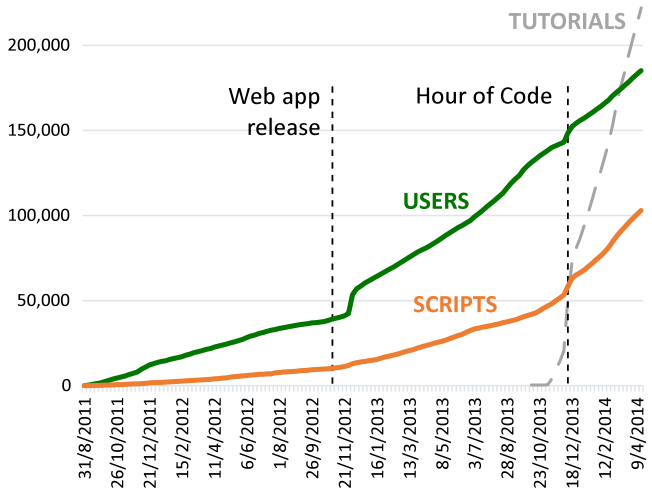

Figure 2: Growth of TouchDevelop

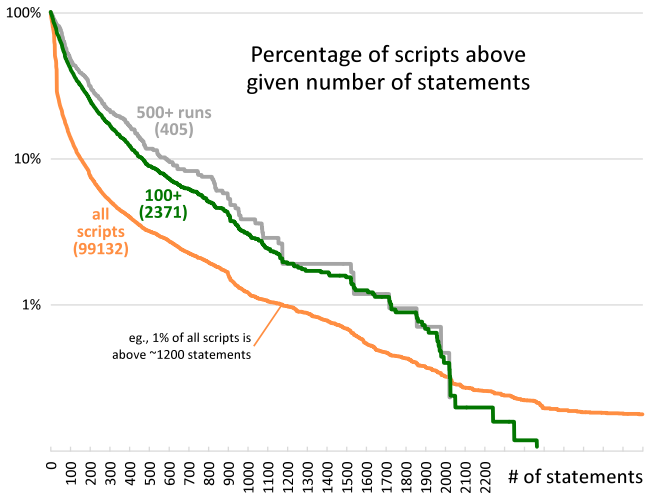

Figure 3: Distribution of script sizes

which often make stack traces non-informative.

We found this functionality useful enough to expose similar information for user scripts: whenever the script crashes, the user can agree to sending crash report to the author. These are then categorized in a similar way.

The analysis of instrumentation data is simpler: it can be visualized over time and categorized by different kinds of devices. We have used it on a number of occasions when deciding to remove unused fragments of user interface code from the IDE.

In addition to automated tracking, we also let users, particularly the ones using "beta" release, report bugs in forms of comments on a specific forum. These can be then categorized, assigned, and tracked (we plan to expose similar functionality for user scripts soon). Given our limited resources, this form of crowd-sourced testing enabled by large TouchDevelop user-base has proven very useful.

4.2 Scripts and users

This section provides basic data about the users of TouchDevelop platform, the scripts they publish, and growth trends. The analysis is based on the public data about users and their published scripts (see § 3.7).

The data-set contains 105,076 scripts published between August 2011, when we first introduced script publishing, and April 2014. We have excluded from further analysis 5,943 scripts published from various system and testing accounts, leaving us with 99,133 scripts.

A *feature* is a built-in function name, a qualified library function name, or a language feature name (*eg.*, "if-statement", "object type definition" or "assignment").

A *feature multiset* for a script contains each feature as many times as it is used in a given script. In particular, the multiset does not contain literals or art references, which are customizable in automatic tutorials (see § 3.5), and are commonly changed during *rebrandings*—when a user clones a script and only changes a bit of text or a picture.

Trivial scripts are ones which have no base script and share the exact feature multiset with at least ten other scripts. These are either very small, or are the result of completing a tutorial.

Overall, 58% (58,116) of all scripts are non-trivial. There are 184,773 users of whom 30,382 have published at least one script, and 14,287 have published at least one non-trivial script.

Figure 2 shows the growth of the TouchDevelop platform over time—published scripts, registered users, and number of started tutorials (scaled down by 50% to fit in the plot). There was a significant bump in number of users (if not scripts) following a round of publicity after the initial release of the web IDE. Similarly, the Hour of Code event brought in quite a few new (but trivial) scripts. The plot also show that growth continues to accelerate.

Figure 3 shows the size of published scripts. While the majority of scripts are small (their median size is 24 statements), 14,031 scripts contain more than 100 statements, and 1,253 script contain more than 1000 statements. Scripts above 5000 statements are outliers—there are only 132 of them—and the biggest script has 9282 statements. Scripts which are successful, measured by the number of runs, are significantly bigger—their median size is 72 statements for scripts with over 100 runs and 96 statements for scripts with over 500 runs.

4.3 Updates and remixes work

The *update size* is the cardinality of the multiset of features used in given script minus the feature multiset of its base (if any). The cardinality of the sum of all update sizes, which is a measure of published edit operations, is 4.0M for trivial scripts and 5.5M for non-trivial scripts. On the other hand, there are 0.8M statements in trivial scripts (which have no base) and 7.7M in non-trivial ones, suggesting that an average non-trivial statement was republished unchanged several times.

Figure 4 shows the update sizes for scripts with no base (*ie.*, initial publications), scripts where the base has the same author (*ie.*, updates), and scripts where the base has a different author (*ie.*, direct remixes). The data is for non-trivial scripts only. The initial publication is by far the biggest, with small incremental updates after that. We later point to quality advantages of scripts with updates. The small sizes of remixes (and their relative low quality, see § 4.7) suggest that they are mostly used as a learning tool, although in some cases the changes are significant (*eg.*,

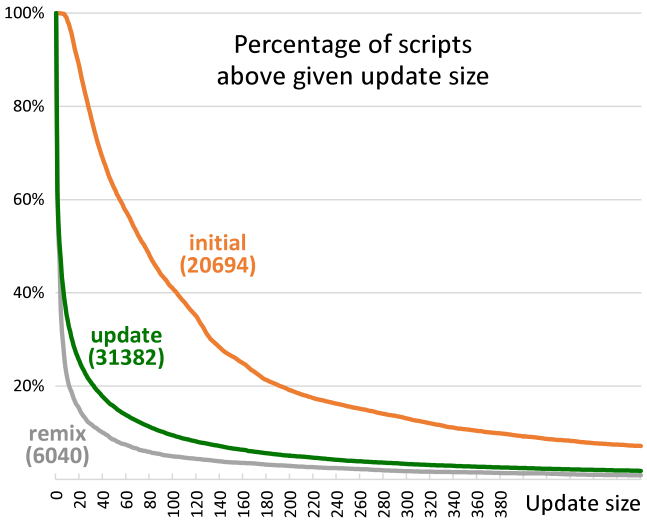

Figure 4: Distribution of update sizes

there are 330 remixes with an update size greater than 100).

Typically, there are at most a few updates published for a given script, however longer update sequences occur: the maximal length of base chain (§ 3.3) is 314, with 890 scripts over 50, 6600 over 10, and 37,422 with a base script at all.

4.4 Script popularity vs. user success

In the remaining sections we want to see how various factor influence user's success in developing apps. Our assumption is that once a user manages to develop an app, they will publish it and it will get run by the author and other users. Of course, the user may not publish, or other users may fail to find (and thus run) the script. However, the number and quality of publications does not suggests users are particularly shy about publishing. We also observe lots of scripts being run by non-authors. We are thus going to take the number of runs of a script as a proxy for user success.

Alternatively, we could use the number of installations of a script or the number of hearts (positive reviews) given to a script. We found these to be correlated, but the number of runs is the most informative, especially for the majority of scripts, which do not have any hearts, but are run a few times.

Profile, coverage, and crash logs also could be used as measures of quality and popularity, but they are often missing and do not capture if the script is performing useful functions.

4.5 Phone is better for development than a PC

Figure 5 shows the mean number of runs and the mean size for scripts published from different kinds of devices. Overall, scripts published from mobile devices (phones, tablets, music players, etc.) are smaller, yet more popular than ones published from desktops (keyboard-equipped desktop and laptop computers). This suggests people are putting more effort into programming from a mobile device.

The good showing of mobile platforms is mainly due to the dedicated TouchDevelop apps for Windows Phone. Our Android app is relatively recent and not yet as feature complete. Tablets are generally also showing good results, comparable with desktop machines.

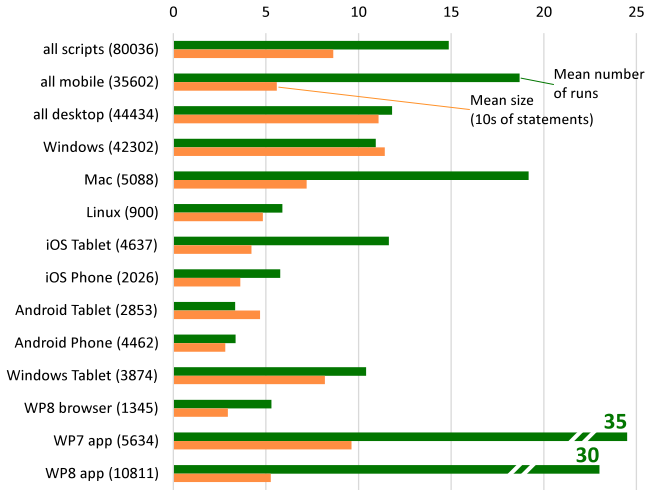

Figure 5: Script popularity by publication platform

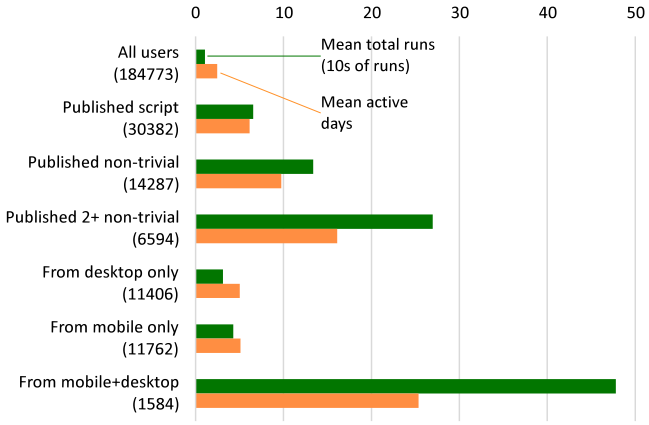

Figure 6: Script popularity by user type

We did not collect data about publication device at the beginning—the plot shows data for the 80% of scrips for which we did collect publication device. Also, Windows tablets are a bit difficult to categorize as they may or may not sport a keyboard, which may or may not be used. They are thus included in both Windows numbers and separately.

4.6 Phone + PC is better yet

Figure 6 shows the total number of runs of all scripts published by a user, averaged over given user set (and divided by 10 to fit in the plot). It also shows the mean number of days the user was active.

Similarly to the data for scripts, users publish higher quality scripts from mobile platforms. In both cases they use the platform for about five days on average. However, users who publish from both desktop and mobile platforms produce much better scripts and use TouchDevelop for almost a month on average. This is encouraging, TouchDevelop is the only platform which currently provides this kind of emerging experience for coding.

4.7 Language features

Figure 7 shows script popularity for scripts using different features of the platform or the language. The data is lim-

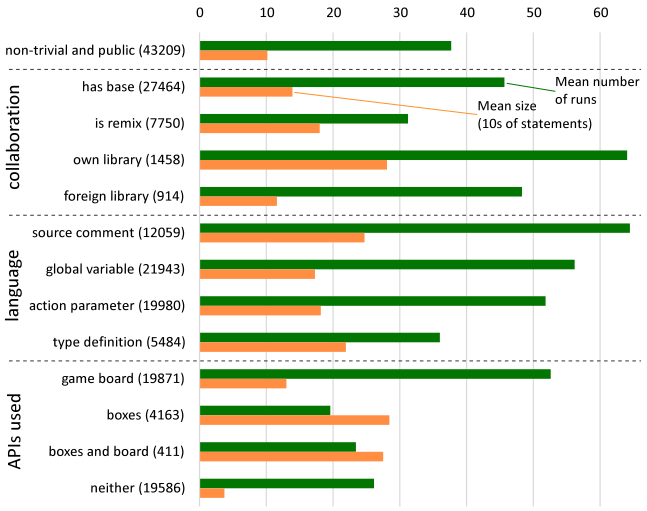

Figure 7: Script popularity by feature used

ited to non-trivial and non-hidden scripts. Hidden scripts account for about 20% of all scripts, but are more common in some of the buckets we analyze (though not in the per-platform buckets from previous sections). Hidden scripts are hardly run by anyone other than they author, so they get artificially low scores.

The first set focuses on different forms of collaboration and version control. Scripts which have a base script are somewhat more popular and bigger than script that do not, which suggests people are finding source control useful. Libraries as a form of modularization (*ie.*, the author of library and the script is the same) seem very successful, whereas usage of other user's libraries is moderately successful. Low popularity of remixes (*ie.*, scripts which have more than one author in the base chain) seem to indicate they are used mostly as learning tool, yet are common.

The second set focuses on the usage of different language features. Scripts with comments in the source seem more popular. Other somewhat advanced features (global variables and functions with arguments) also have positive influence on popularity. However, users do not seem to know how to handle more advanced features, like type definitions and boxes [5] (see below).

The last set splits the scripts by the main API used. Scripts that use the game board (which includes physics and 2D graphics engines) are most numerous and popular. Boxes are TouchDevelop's way of constructing more complex UIs. They are an advanced feature, which users seem to find hard to master. Finally, it is possible to create very simple scripts (note the low average size) with simple UI, which use neither game board or boxes.

The relative run counts of games and apps may be due to users being more inclined to play games than use productivity apps. However, even the gap between apps with and without boxes is significant.

5. RELATED WORK

Many aspects of our CIDRE definition (§2) can be found in lesser combination in existing projects and websites; to the best of our knowledge, their full combination is unique to the TouchDevelop project.

5.1 Web-based Software Development

Community websites such as StackOverflow demonstrate how to encourage programmers to share knowledge, and online repositories such as GitHub combine cloud storage, social functions to encourage collaboration, outsourcing functions like build to other web services. However, neither one supports both the development activity and the deployment of applications.

Providing the functionality of traditional IDEs as a website is becoming increasingly popular, and there are many web-based IDEs available today, such as `codenvy.com`, `koding.com`, or `visualstudio.com`. While they provide typical IDE functionality (develop, compile, test, debug) and parts of the emerging experience (ubiquituous workspace, secure identity, and code repository), they fall short on the many other aspects of our CIDRE vision.

There is no deep integration of social features to support the online community and foster open collaboration. Also, support for tracking the whole life cycle of applications (including deployment and telemetry) is only slowly emerging, and typically limited to the parts of the application that execute in the cloud. Almost all web-based IDEs still still do not support offline operation. Finally, as far as we have seen, web-based IDEs all but ignore the mobile experience and still squarely focus on programming from the traditional desktop with large screens, keyboards, and text-based programming languages.

Abandoning traditional programming methodology in favor of high-level programming abstractions and building blocks that simplify the development of applications also is seeing more uptake. A good example are online app-creation wizards [1, 2]. However, TouchDevelop remains unique in this space as its design is still *code-first*, allowing the development of complete programs, as opposed to code fragments in a static template.

5.2 Development for Non-expert Programmers on Mobile

PocketCode [12] is a tool similar to the initial version of TouchDevelop—it focuses on single-user programming directly on a phone or tablet. The programming language is far simpler and its main aim is education.

AppInventor [9] lets users create Android apps by drag and dropping blocks representing various ASTs on a separate PC. The programming language is simpler than in TouchDevelop (for example lacking library abstractions), and the cloud and social support is more limited. On the other hand, the support for Android-specific APIs and UI is superior in AppInventor. Interestingly, the code editor is fully structured—expressions are also edited as trees. The effectiveness of the two approaches was recently compared [8].

6. FUTURE WORK AND CONCLUSIONS

TouchDevelop's main use so far has been in the educational context—as an introductory platform for mobile, or even general, development. Yet, we have also seen some professional use, for example the MindSticks game, ranking very high in the Windows App Store. In future, we plan to push the semi-professional angle more by enabling features like server-side execution (with full programming language integration), offline AST-based merge functionality in version control, real-time collaborative editing of code, and inte-

grated bug-tracking. This should provide us with more data how a CIDRE would be used by more advanced developers.

We also plan to experiment, or encourage experimentation by others via the cloud plugin mechanism, with various program analyses to enable faster learning and better code quality for non-expert programmers.

We believe that TouchDevelop integrates many of the features which will become more common in the IDEs of the future. It also represents a convenient vehicle for programming language research. We encourage the reader to try the IDE (at `touchdevelop.com`) for themselves, and invite them to extend it.

7. REFERENCES

[1] 10 Excellent Platforms for Building Mobile Apps. `http://mashable.com/2013/12/03/build-mobile-apps/`.

[2] Microsoft Project Siena. `http://microsoft.com/projectsiena`.

[3] TypeScript Langauge Website. `http://www.typescriptlang.org/`.

[4] S. Apel, O. Leßenich, and C. Lengauer. Structured merge with auto-tuning: balancing precision and performance. In *ASE*, pages 120–129, 2012.

[5] S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in ui programming. In *PLDI*, pages 95–104, 2013.

[6] S. Burckhardt, M. Fähndrich, D. Leijen, and B. Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 7313 of *LNCS*, pages 283–307. Springer, 2012.

[7] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, pages 67–86, 2012.

[8] A. Harzl, V. Krnjic, F. Schreiner, and W. Slany. Comparing purely visual with hybrid visual/textual manipulation of complex formula on smartphones. In *DMS*, pages 198–201, 2013.

[9] J. Liu, C.-H. Lin, P. Potter, E. P. Hasson, Z. D. Barnett, and M. Singleton. Going mobile with app inventor for android: a one-week computing workshop for k-12 teachers. In *SIGCSE*, pages 433–438, 2013.

[10] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10:16:1–16:15, November 2010.

[11] A. Puder, N. Tillmann, and M. Moskal. Exposing native device APIs to web apps. In *Proceedings of First ACM International Conference on Mobile Software Engineering and Systems*, MobileSoft 2014, 2014.

[12] W. Slany. A mobile visual programming system for android smartphones and tablets. In *VL/HCC*, pages 265–266, 2012.

[13] T. Teitelbaum and T. Reps. The cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, Sept. 1981.

[14] 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, 2011.