New Opportunities for Extracting Insights from Cloud Based IDEs

Yi Wang[1], Patrick Wagstrom[2], Evelyn Duesterwald[2], David Redmiles[1]
[1] Department of Informatics, University of California, Irvine, CA 92697
[2] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
[1] {yiw, redmiles}@ics.uci.edu
[2] {pwagstro, duester}@us.ibm.com

ABSTRACT

Traditional integrated development environments (IDEs) provide developers with robust environments for writing, testing, debugging, and deploying code. As the world becomes increasingly networked and more services are delivered via the cloud, it is only natural that the functionality of IDEs be delivered via the cloud. In addition to simplifying the provisioning and deployment of new IDE features, and making it easier to integrate with other web native tools, cloud based IDEs provide some fundamental advantages when it comes to understanding the behavior of a wide community of software developers. One of these advantages for the IDE provider is the ability to transparently monitor and analyze the real-time fine-grained actions of a large number of developers. In this paper, we explore how to leverage these transparent monitoring capabilities of cloud based IDEs to develop advanced analytics to understand developers' behavior and infer their characteristics. We demonstrate the feasibility of this research direction with a preliminary study focusing on the way that source code files grow for different developers, development tasks, and skill levels. We then analyze the trends of source code file growth and find growth is more similar within subjects than within tasks.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Human Factors, Measurement

Keywords

Cloud based IDE, fine-grained data, analytics, JazzHub

1. INTRODUCTION

The era of cloud computing is upon us. According to the 2013 Eclipse Community Survey[1], 53% developers have developed or deployed cloud based applications, more than doubling the 2012 rate of only 21%. However, to this point much of the attention on cloud based applications has focused on deployment, execution, and management of applications, sometimes called devops. With nearly ubiquitous access to the internet, and the variety of devices, laptops, tablets, and phones, that can support rich applications, we assert that the next mainstream software development environments will be cloud based IDEs, such as JazzHub [7].

These cloud based IDEs provide a number of advantages, as they are web native they can take advantage of the rich opportunities afforded by the client-server model of web applications. These advantages include for example, automatic saving and backup of code, real time pair programming across a distance, and the ability to harness a huge number of backend computers to perform tasks, such as build and search, that previously were done on the desktop. These IDEs make the development process truly scalable and flexible: no installation on the desktop is required, potential collaborations among unlimited numbers of developers is enabled, and maintenance is greatly simplified as new versions can be automatically provisioned to all users.

Cloud based IDEs also provide another great advantage for developers, managers, tool builders, and researchers alike: they provide a centralized point for the transparent collection of data related to the software development process. Thanks to improvements in JavaScript and underlying communications protocols such as WebSockets, which allows for every keystroke on a web page to be easily sent back to a server, and WebRTC, which allows for video and audio to be sent directly from the browser to the server, it's possible to see, analyze, and perhaps even understand every action that users take while developing software. This unprecedented transparency enables us to extract insights about the behavior of the IDE's developer community that can be exploited in many new and novel ways. Developers may be provided with relevant and real-time assistance based on not only their own behavior but also based on leveraging the experience of the entire developer population. Development managers may gain new insights about individual and team contribution and finally, the IDE may leverage real-time insights about actual tool and feature usage and user experience to improve future versions of the IDE.

This paper presents an example of how to develop advanced analytics by leveraging the transparency provided by cloud based IDEs, which can be further used to understand developers' behavior and infer their characteristics. We developed a concrete example to demonstrate the feasibility of this approach. In this example, we focused on a specific type of fine-grained data, the way that developers add and remove individual characters to and from source code, which usually cannot be easily measured and recorded at this fine-grained level in a transparent fashion for large numbers of developers. After collecting data in a controlled experiment with IBM JazzHub[1], we used time series analysis to study the data. The analyses reveal a set of interesting results, e.g., source code growth traces are significantly more similar within the same developer than the same task. Implications, limitations and future work is also discussed.

2. NEW OPPORTUNITIES

2.1 Prior Work

Collecting a developer's actual sequence of user interactions with development tools is crucial for understanding developer behavior as well as task characteristics. A record of actual user interactions can be used to create a "work context" of the user's behavior. These work contexts enable tool designers to provide more relevant support by providing the information users need at their fingertips. Early work in this area by Murphy et al. investigated how different Eclipse features were used by Java developers [9]. Their work was extended from monitoring feature usage to constructing task contexts for making recommendations [5, 6] and identifying developer attributes [4]. Other work developed metrics from monitoring programming activities in order to predict software defects [8]. There has been a large body of similar work but almost all of these studies require access to the client to install additional instrumentation in the IDE, and the analyses are typically based on coarse-grained, high-level actions (e.g. "use refactoring function", "read code"). The lack of transparency in the data collection may influence a developer's natural behavior while the focus on coarse-grained information may limit the kind of analytics that can be applied. Cloud based IDEs bypass both of these problems by enabling transparent collection of fine-grained user data.

2.2 Advantages of Monitoring Cloud Based IDEs

Cloud based IDEs fundamentally change the way we can collect data about developer behavior as all user interactions happen inside the web browser. We can passively monitor server-side, by tracking the communication between client and server using techniques such as IBM TeaLeaf. In contrast to IDE plug-in style activity monitors, such as Mylyn [6], the data capturing does not require any client-side installation or instrumentation. As a result, instrumentation and monitoring is less-invasive to normal user behavior. As data collection happens in the background and data are processed on a remote server, developers are usually unaware of the fact that their interactions with the IDE are captured.

Another advantage of moving the monitoring control away from the client and to the server is that monitoring has several can easily be scaled up to large (in fact unlimited) numbers of users. Monitoring efforts are no longer limited to the select, and possibly biased, subsets of the IDE user population whose IDE clients can be instrumented. The user population of a cloud based IDE can become subject to monitoring resulting in more diverse and more broadly representative data. A proposed architecture for this style of monitoring can be seen in figure 1.

[1]JazzHub: https://hub.jazz.net/

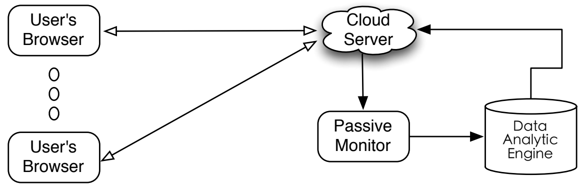

Figure 1: Mass monitoring in a cloud based IDE.

2.3 New Possibilities in Cloud Based IDEs

The ability to transparently monitor large classes of users, and even entire user population opens up new possibilities for deep analytics. Large scale monitoring yields large representative data sets that can be analyzed statistically for the purposes of classifying and possibly even predicting behavior in a user community. Furthermore, cloud based IDE monitoring can leverage the wisdom of the crowd to provide more advanced services to the IDE user. Consider the example of intelligent code completion. Using a traditional IDE, intelligent code completion systems can record the details of a developer's activities and use the collected data to make recommendations. While this method can make use of an individual's experience, it fails to utilize the cross-experience of a large number of individuals. In a cloud based setting intelligent code completion systems can have a server component that passively captures data of not only a single developer but of thousands of developers. This can fundamentally change how intelligent code completion systems work. They can now include advanced analytic techniques to leverage the experiences of mass populations [2]. For example, time series analyses could be used for identifying sequence patterns, or Bayesian inference for predicting developers' future behaviors, etc.

Deep analyses of IDE user behavior can also provide valuable input to the IDE providers to continuously improve the IDE based on an actual user behavior feedback. Data on overall feature usage, which features are used together, and the profiles of users most likely to use each feature can be easily collected in a cloud based IDE. This allows IDE providers to quickly the impact of feature adoption and to be more responsive to user wants and desires.

3. CASE STUDY: UTILIZING FINE-GRAIN -ED SOURCE CODE GROWTH DATA

We designed a controlled laboratory experiment as a proof-of-concept to better understand the potential of transparent fine-grained behavior monitoring as made possible by cloud based IDEs. In this study, we worked with JazzHub to observe and collect fine-grained source code growth data from a small pool of software developers. Users were given a series of simple programming tasks and asked to develop code in the editor integrated into JazzHub. We continuously monitored the current size of the source code in the active editor window by capturing all communication between the browser and the JazzHub server while users were performing these tasks. Source code growth could then be reconstructed from the series of auto-save and explicit save events in the browser/server communication. We analyzed the resulting source code growth data with advanced data analysis techniques to generate insights about the characteristics of different software developers, and their work patterns.

3.1 User Study Design

Subjects & Process. Our study used eight subjects who were all IBM employees at the time of our study (seven were summer interns, one was a recent hire). Although three programming languages (JavaScript, PHP, and Python) were offered, all subjects chose Python. Using a specially instrumented laptop that recorded all network connections and the screen during the study, subjects used JazzHub to both write and execute code. Each subject was asked to solve up to eight problems, arranged in increasing order of difficulty. The subjects were instructed to aim for completing as many programming tasks as possible in the allocated time.

Evaluation. The performance of each subject was evaluated and ranked. If a programming problem was solved, the subject would receive basic credit for the problem. Bonuses or penalties would be assigned if there were improvements (such as a solution with superior runtime) or minor deficiencies (such as not handling corner cases or excessive runtime). The best performer (subject 6) finished all 8 tasks, while the worst performer (subject 4) only finished 3 tasks.

Extracting Code Growth Data. Source code growth data was extracted from the saved HTTP conversations. We enabled the JazzHub "autosave" function in the source code editors, so that the browser would periodically use the `PUT` method to update the server-side copy of the current source file with the contents of the local web browser (approximately every 15 seconds during active editing of the file). The `PUT` method's headers contains the information of the exact time of the `PUT` invocation and the length of the source code file at that time. We extracted this information for each task and represented the task's source code growth traces as a series of tuples $< time, length >$. The traces for different subjects vary in length depending on the amount of autosave events and overall time spent on a task.

3.2 Analysis, Results, and Findings

3.2.1 *What Does Code Growth Look Like?*

When studying the code growth traces of individual subjects, it is immediately apparent that code growth behavior varies significantly between subjects, and even from task to task. Our analysis can examine differences in more detail. Figure 2 shows a comparison between the traces for the first three programming tasks collected from the worst performer (subject 4) and the best performer (subject 6). There are pronounced differences in the overall length of the individual code growth plots between the subjects (note the difference in the scale). It is easy to see that there are significant differences in the shapes of these source code growth traces. Subject 6's source traces are very smooth and generally monotonically increasing. In contrast, subject 4's traces are erratic; with many quick increases, some drops, and long plateaus. These shape patterns can help us gain insights into the developer's experience while they were programming. Drastic drops imply code deletion indicating that corrections or refactoring occurred. Plateaus suggest "think time" possibly because users were experiencing challenges preventing them from making steady progress.

Based on a shape analysis of the collected code growth traces in real-time, we may be able to provide live assistance to developers according to the inference of their current states. By computing the statistics of these patterns, we may further identify characteristics of developers. In the next section we look at one such mechanism to statistically evaluate code traces.

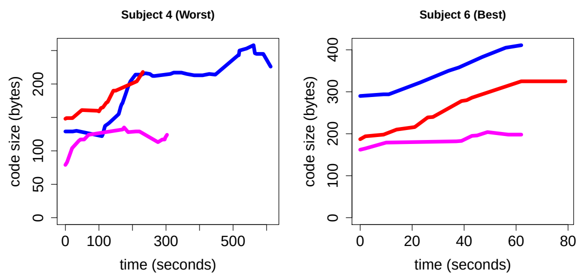

Figure 2: The traces of the worst and the best performer. We only plot the best performer's first 3 traces. Note: the scales of two plots are different.

3.2.2 *Cointegration Analysis*

Cointegration is a time series analysis technique frequently used in quantitative finance. Cointegration is better than correlation as an indicator of how similar the shapes of two time series are because correlation usually fails to indicate shape similarities, especially, when these similarities are staggered in time. Intuitively, if two time series are integrated, they are unlikely to diverge over time [3]. Hence, we can use it here to test the degree to which two different traces have a similar general shape. In order to perform cointegration analysis, we first normalized the original traces to equal length traces. We found that most traces had between 10 and 25 recorded samples, so we normalized each trace into new 15-sample traces and used the differential value to replace the absolute value. This way, we obtained 41 new traces that start from 0 and have 15 values each. For each pair of normalized traces, cointegration was tested using the Engle-Granger method [3], this generates a p value for each pair of traces that represents the probability that two traces are cointegrated. The analyses resulted in 820 statistics, which can be represented by the lower triangle of a 41×41 matrix. If using $p = 0.05$ as the critical value, there are a total of 217 cointegrated pairs.

3.2.3 *Cointegration Network*

In order to provide an intuitive view of the results, we encoded the cointegration statistics into 0 or 1 according to their significance. Thus, the cointegration matrix can be viewed as an adjacency matrix of a network, as shown in figure 3). In this network, each vertex represents a trace, and there is an edge if two traces are cointegrated. In figure 3, a label `Sm.n` represents subject ID `m` working on task `n`.

This network leads to several observations. First, an interesting phenomenon is that the "best" developers (subjects 6 and 8, whose traces are represented by white dots in figure 3) share fewer connections than the average. There are only 6 edges between them although they contribute 15 traces (5.71% of all possible edges), while the other six subjects have 112 inter-subject edges for 26 traces (34.46 % of all possible edges). They are mostly located in the peripheral part of the network of Fruchterman-Reingold layout. Their positions actually indicate the shape of these traces are more "unique" than the overall population of traces.

Given the small number of our sample, we cannot formally establish this argument. But this preliminary results

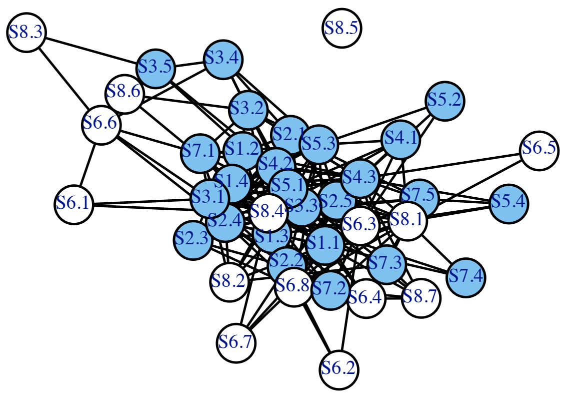

Figure 3: The cointegration network.

may suggest that the "best" or most experienced developers might have more strategies in solving programming problems, which causes the traces they create to be more diverse. In contrast, less experienced developers may follow only a limited number of similar strategies to solve their problems, making their traces more similar to each other. This can be potentially used as an indicator in inferring developers' skill level. Another interesting finding is that there is some kind of "lag" effect between the traces of experienced developers and trace of developers with less proficiency. E.g., `S6.4` connects with several other subjects' traces of the first task. The tasks were ordered by increasing difficulty so that a more experienced developer may subjectively perceive the difficulty of a higher numbered task similarly to how a less experienced developers perceives the difficulty of the first (easiest) task. The connections between `S6.4` and several first task traces from weaker performers suggest such a similarity in subjective perceived difficulty.

3.2.4 *Task Differences vs. Individual Differences*

To understand to what extent we can characterize the behavior of users from code growth traces we would like to determine whether code growth traces are more similar for a collection of users doing the same task, or for an individual user doing a collection of different tasks. To answer this question, we compute the density of edges in three categories: I: (subject-subject), II: (task-task), III: (task-subject) for the network in figure 3. The results are: 1. Category I: 0.3085; 2. Category II: 0.2479; 3. Category III: 0.2611. Obviously, the density of subject-subject edges is higher. We further tested whether the differences are statistically significant using Kruskal-Wallis test. Edges in category I are significantly more probable than the other two ($p_{(I,II)} < 0.01; p_{(I,III)} = 0.03$). There is no significant difference between II and III. These findings indicate that the shape a development trace is more unique to an individual developer than it is the to the task at hand.

4. DISCUSSION AND FUTURE WORK

In this paper we've touched on some of the advanced possibilities of observing and analyzing developer behavior in cloud based IDEs. We intend to continue to investigate fine-grained actions of developers that can be extract and analyzed in these environments. In particular, we are optimistic that increased adoption of cloud based IDEs for traditional development tasks and the shift in the focal point used for studying developer activity that this enables, as mentioned in section 2.3, will lead to a plethora of future analysis techniques. This shift provides unprecedented transparency of the mass population's development activities. Thus, the scalability and reach of theses kinds of analyses will be greatly improved. Given access to a suitable community and with suitable permissions we can now monitor thousands of developers, rather than being confined to traditional studies of a handful of developers.

Possibly even more exciting is the fact that the centralized collection of this data allows information collected from one user to be easily leveraged by other users in near-real time. For example, consider an environment that is aware of the methods used by all developers. Intelligent code completion options (e.g. intellisense) could be tuned to provide only the most likely options across all users or across the subset of users that bear substantial similarity to the current developer. This can be furthered by the fact that even complex calculations to provide assistance to the user can be transparently transferred to other servers without the user experiencing a decrease in performance.

This paper presents only a glimpse into the insights we expect to gain about how developers work in cloud based IDEs. As the community around JazzHub and other cloud based IDEs grows we look forward to continuing our analysis to get a better understanding of how developers work in the real world and how the centralized collection and transparency enabled by cloud based development environments can provide the next generation of rich and productive environments for software development professionals.

5. ACKNOWLEDGEMENTS

This work is partially supported by NSF grant 1111446.

6. REFERENCES

[1] Eclipse community survey (2013). http://wp.me/p1HAa-H9. Accessed: 2013-11-19.

[2] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini. IDE 2.0: collective intelligence in software development. In *Proc. FoSER 2010*, pages 53–58, 2010.

[3] E. Chan. *Quantitative Trading: How to Build Your Own Algorithmic Trading Business*. Wiley, 2008.

[4] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proc. ESEC/FSE 2007*, pages 341–350, 2007.

[5] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. FSE 2006*, pages 1–11, 2006.

[6] M. Kersten and G. C. Murphy. Task context for knowledge workers. In *Proc. AAAI 2012 Activity Context Representation workshop*, 2012.

[7] J. Lautamäki, A. Nieminen, J. Koskinen, et al. Cored: Browser-based collaborative real-time editor for java web applications. In *Proc. CSCW 2012*, pages 1307–1316, 2012.

[8] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proc. ESEC/FSE 2011*, pages 311–321, 2011.

[9] G. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.