Niche vs. Breadth: Calculating Expertise over Time through a Fine-Grained Analysis

Jose Ricardo da Silva Junior
Esteban Clua, Leonardo Murta
Universidade Federal Fluminense
Niterói, Brazil
{jricardo,esteban,leomurta}@ic.uff.br

Anita Sarma
Computer Science and Engineering Department
University of Nebraska, Lincoln
Lincoln, United States
asarma@cse.unl.edu

***Abstract*—Identifying expertise in a project is essential for task allocation, knowledge dissemination, and risk management, among other activities. However, keeping a detailed record of such expertise at class and method levels is cumbersome due to project size, evolution, and team turnover. Existing approaches that automate this task have limitations in terms of the number and granularity of elements that can be analyzed and the analysis timeframe. In this paper, we introduce a novel technique to identify expertise for a given project, package, file, class, or method by considering not only the total number of edits that a developer has made, but also the spread of their changes in an artifact over time, and thereby the breadth of their expertise. We use Dominoes – our GPU-based approach for exploratory repository analysis – for expertise identification over any given granularity and time period with a short processing time. We evaluated our approach through Apache Derby and observed that granularity and time can have significant influence on expertise identification.**

Keywords—expertise identification; exploratory data analysis; GPU computing

I. INTRODUCTION

Identifying expertise in a software project is an important issue for task allocation, personnel hire, onboarding, and development help, among other activities. It has been observed that when stuck in a task, developers often use their implicit knowledge of work dependencies to identify a developer who can help [1], or rely on their social network to find others who might know enough about the artifact in question [2]. In fact, managers often use informal processes to facilitate their team members to come talk to them (e.g., a manager keeping a candy bowl in his office), so that they are aware of who is having what kinds of problems and can direct developers to each other. However, the large scale of software development combined with developer turnover can make such informal processes in identifying expertise difficult [3]. In globally distributed software development (or in the case of open source development), finding an expert is even more challenging.

Software development leaves traces of development activities in the repository, which can then be used for inferring the expertise of developers. Existing approaches in expertise identification have used machine learning to identify expertise among developers based on their edit histories. These approaches have been used for supporting automated bug triaging (to find the appropriate developer to perform a bug fix [4]) or support collaboration in a team by mining relevant artifacts to a given change request, and recommending developers based on their source code changes, experience and contributions [3].

However, these approaches can be inefficient when large scale of data needs to be processed. A large software project may comprise thousands of files, with hundreds of developers making thousands of commits per month, making it difficult or even impossible to process this data at interactive rates. Current approaches often work off-line and scope the analysis to manage scalability, which may lead to inaccuracies in the results. Some of the common strategies for scoping the analysis are: (1) filtering the data, (2) performing coarse-grained analysis, and (3) overlooking evolution.

In the first case, available tools either scope the amount of data that is processed or the time period over which processing is performed. For example, EEL [5] scopes the analysis to 1,000 project elements when identifying expertise in a team, thereby restricting the application to smaller chunks of data. In the second case, tools often analyze data at a coarse-grain, such as the file level [6]. The problem of performing analysis at this level is that a developer may be recommended as an expert of the whole file, even if she only intensively worked on a small portion of that file. Analysis at the finer-grain (method or lines-of-code), however, leads to scale issues. Finally, most current approaches, such as Expertise Recommender (ER) [7], consider the entire history of the project at once to recommend experts, overlooking the fact that artifacts evolve over time and that developers may change their roles. Further, temporal analysis can show how expertise of a development team changes and whether there are artifacts that lack experts at a given moment in time.

In this paper, we propose a novel approach for identifying expertise that considers not only the total number of edits over a given artifact, but also the spread of the change over the parts of it, and the time period when the change was performed.

We analyze expertise by considering fine-grained changes and time frames. We organize fine-grained data extracted from software repositories into multiple matrices. For example, by extracting the lines of edited code in a file (code churn) from the version control repository we can reverse engineer the information to create a matrix of methods that were added, removed, or changed as part of each change set ([commit|method] matrix). Similarly, we can also create a matrix of developers that were responsible for each commit ([developer|commit] matrix). By analyzing the project we can create matrices that represent the composition of artifacts. For exam-

We thank CNPq, CAPES, FAPERJ, and NSF (through awards: IIS-1110916, IIS-1314365, and CCF-1253786) for the financial support.

ple, which methods belong to which class, which classes belong to a package, and so on. These composition dependencies are then operationalized into matrices ([class|method], [file|class], and [package|class]). Operating over these matrices allows us to identify developers who are experts on an artifact at different levels. We can identify experts at a fine-grained level (e.g., methods), or at a coarser-grained level (e.g., classes, files, packages, or the project as a whole).

In addition, we also analyze the breadth of changes in an artifact. That is, we can differentiate if a developer has expertise on only a specific part of an artifact (e.g., a single method in a file) or has a broader experience with the artifact (e.g., developer has edited the majority of the files). Finally, we allow users to identify the time period over which expertise is to be calculated, as well as analyze how expertise is changing for a project element over time.

We use Dominoes [8] – our GPU-based approach for efficient, large-scale repository analysis – to perform our analyses. The parallel architecture of Graphics Processing Units (GPU) adopted by Dominoes allows it to process the underlying matrices much faster than what can be possible with CPU processing [9]. This allows us to analyze expertise as an interactive analysis process, even for large datasets.

We evaluated our expertise identification approach in the Apache Derby[1] project. We chose Apache Derby since it is stable project, has lots of activity since the beginning, and is a long-lived project. When we contrast our expertise identification technique with the usual (code) edit based approach, we observed that our breadth-based analysis at fine-grain yields different expertise results in about 28% of the project files (about 977 files). We also show how developer expertise in the Apache Derby project fluctuates over time. Finally, by running these analyses in GPU, we observed a boost in performance when compared to CPU processing.

II. Breadth-Based Expertise Identification Over Time

Expertise identification has typically considered the total number of edits to an artifact that a developer has made, without considering the location of these edits. Here, we introduce our approach for identifying the breadth of expertise of a developer in a project, and over time. This approach considers two important factors pertaining to expertise: granularity of analysis and time. In essence, artifacts are not atomic, and local expertise (in specific parts of an artifact) should be differentiated from global expertise (in the whole artifact). Moreover, as artifacts naturally evolve over time, the expertise of a developer diminishes unless she has kept familiarity with the artifacts over time through her development efforts.

A. Granuarity Matters

Here we discuss two different strategies for calculating the expertise of a developer in a given artifact (e.g., file): Expertise of a Developer (ED) and Expertise Breadth of a Developer (EBD). The first one considers the entire artifact as an atomic element, and is vastly adopted in the literature [5], [10], [11]. However, to differentiate between developer expertise, we use the Z-score in our analysis. The second approach uses the underlying composition structure of the artifact (e.g., methods) to perform a fine-grained analysis when calculating expertise.

Expertise of a Developer (ED): it identifies the frequency of changes to an artifact (e.g., file, project, etc.) by a given developer. Frequency of edits has long been used as a proxy for identifying the knowledge that a developer has about an artifact, typically a file [12]. The intuition is that the more someone has edited a file, the more working knowledge that person has about that file. The frequency of edits can therefore help answer two related questions: (1) who is the developer that is an expert for a given file?, and (2) which files is a given developer an expert of?

Table I presents a scenario with three developers who have worked on three files ([developer|file] matrix – DF for short). Note we arrive at the DF matrix by operating over the basic tiles ([developer|commit] x [commit|file]). The cells in the derived matrix DF represent the number of times a developer d_i edited a file f_j. Besides that, Table I also shows the number of commits performed by each developer (note that it is different from summing all columns in a row, as a commit could comprise more than one file).

TABLE I. DEVELOPER X FILE

Project	*FileA.java*	*FileB.java*	*FileC.java*	*Total Commits*
Alice	14	2	20	28
Carlos	10	24	12	25
Bob	25	10	8	40

To answer the first expertise question (who is an expert for a given file f_j), we search for the developers who edited the file the most. This is done by scanning down the column of f_j in the DF matrix. In our simplistic example (see Table I), if we want to identify an expert for *FileC.java*, we would indicate Alice. Carlos would be considered as the second most knowledgeable developer in that file.

To answer our second question, about the expertise of a specific developer d_i, we scan the rows in the matrix for the highest values. In our example, we find that Alice has expertise in *FileC.java*, Carlos in *FileB.java*, and Bob in *FileA.java*.

A key challenge in this approach is identifying the right threshold to use. For example, is there a minimum number of changes that a developer must have performed before they can be considered as an expert? Further, if two developers have changed the file, how do we determine who can be defined as "the" expert – should that be the person with the most edits? For example, if we are going to compare the expertise of Alice versus Carlos on *FileA.java*, because Alice has made four more changes than Carlos, does Alice clearly have more expertise than Carlos? Does a difference of four additional edits matter, or should there be a minimum distance between the numbers of edits to differentiate expertise among developers?

To overcome this challenge, we applied the ***Standard (Z) Score*** [13] to statistically identify the appropriate thresholds. We convert the absolute scores (support) into Z-scores according to Equation 1. In this specific case, x is the absolute score

[1] Derby Repository: https://github.com/apache/derby

in Table I and μ and σ are the mean and standard deviation of the number of edits for each file, respectively.

$$z = \frac{(x - \mu)}{\sigma} \quad (1)$$

When using standard (Z) scores (see Table II), cells above zero indicate values that are above the mean, that is, they indicate that a developer has changed that file more than the mean number of times that file has been changed in the project. Similarly, cells below zero indicate values below the mean. Moreover, cells above (or below) one, two, or three indicate values above (or below) one, two, or three standard deviations from the mean, respectively. For example, when we consider *FileC.java*, we see that it has been changed on average 13.33 times (summing column 3 in Table I and dividing by the total number of developers in the project), and that Alice has edited the file 1.34 standard deviations above the mean (Table II, cell value for Alice's edits to *FileC.java*).

TABLE II. STANDARD SCORE

Project	*FileA.java*	*FileB.java*	*FileC.java*
Alice	-0.37	-1.10	1.34
Carlos	-1.00	1.32	-0.27
Bob	1.37	-0.22	-1.07

We assume zero as the threshold for determining expertise. That is, to be counted as an expert of a file a developer must have edited more than the mean change rate of that file. In our example, this measure enables us to quickly identify the expert developer for each of the files. Alice can be considered an expert in *FileC.java*, since she has made significantly more changes than any other developer in the project. Similarly, Bob is an expert in *FileA.java*, and Carlos in *FileB.java*. To be considered a higher expert than another, we require that the developer has to have edits that are at least one standard deviation higher.

Expertise Breadth of a Developer (EBD): Here we challenge the assumption that a developer who has more commits to a file has knowledge of the (entire) file. Normally, a file comprises of a set of classes and methods and it is possible that a developer only performed niche changes to a subset of methods or internal classes. In such cases, numerous edits to only a subpart of the file does not guarantee that the developer is an expert on the entire file. We, therefore, analyze changes at different levels of granularity to create a more nuanced understanding of expertise.

To calculate EBD we run an analysis at a fine grain. In our case study, we analyzed edits at the method level (however, in our approach it is also possible to analyze expertise at the lines-of-code level, if needed). We first calculate the absolute scores (count of number of edits to the [developer|method] matrix – DM for short – which is computed as [developer|commit] x [commit|methods]). We then transform the absolute scores into standard (Z) score. Finally, using zero as a threshold, we count the cells that have positive numbers as a measure of expertise.

As an example, consider Table III (DM matrix), which is composed of methods present in *FileC.java* and the number of commits performed by each developer, which involved those methods. Table IV presents the Z-score calculated from Table III. In this example, it is possible to see that even though Alice has edited *FileC.java* the most (20 times) and therefore had the highest ED for this file (Table I), most change she has made were only to a single method in the file. In contrast, Carlos has made only 12 changes (as compared to 20 by Alice), but his changes are spread across all the methods in the file. Therefore, based on our proposed metric, Carlos has a higher EBD in *FileC.java* (in Table IV Carlos has positive entries for three methods). If we now look at the expertise of developers in the project that comprises these three files, we find that Carlos is the expert in *FileC.java* and Alice in *FileA.java*. Carlos and Bob both have expertise in *FileB.java*, although Carlos has a higher expertise (see Table V, which presents experts by each file as well as its Z-score in parentheses; we use 0 again as threshold when determining expertise).

TABLE III. DEVELOPER X METHODS (DM MATRIX)

FileC	*a()*	*b()*	*c()*	*d()*
Alice	2	1	1	16
Carlos	4	3	4	1
Bob	2	1	3	2

TABLE IV. DEVELOPER X METHODS Z-SCORE AND EBD_{METHOD}

FileC	*a()*	*b()*	*c()*	*d()*
Alice	-0.71	-0.71	-1.34	1.41
Carlos	1.41	1.41	1.07	-0.78
Bob	-0.71	-0.71	0.26	-0.63

Finally, calculating EBD for the project as a whole when considering the method-grain based approach is shown in Table V. Note that here we omit presenting how the Z-score at method level for *FileA.java* and *FileB.java* has been derived because of space constraints. In Table V we find that Carlos has the highest expertise in the project (he has two cells that have positive Z-scores in different columns) when compared to Alice and Bob (both have only one cell with positive Z-score). Therefore, EBD can serve as a more precise measure of the extent of knowledge of a developer when fine-grained edit data is available.

TABLE V. EXPERTISE AND Z-SCORE AT FILE LEVEL

Project	*FileA.java*	*FileB.java*	*FileC.java*
Alice	3 (1.41)	0 (-1.34)	1 (-0.71)
Carlos	1 (-0.71)	3 (1.07)	3 (1.41)
Bob	1 (-0.71)	2 (0.27)	1 (-0.71)

Our approach can perform (expertise) analysis at different levels of granularity by using the appropriate composition matrix. Here we discussed how we can identify expertise of developers for the project by recursively considering edits from the method level (EBD^M) to the file level and then to the project level. We can follow a similar strategy to identify experts for the project by analyzing edits directly from the file level (EBD^F). The only difference is that the latter considers files as atomic elements and does not take into account location of edits. That is, it computes the EBD of the whole project based on the ED of each file by considering the positive Z-score values.

Considering our example, if we were going to simply use ED for determining expertise for the project, that is, the person who has made the most changes to the project as the expert, then Bob would be considered the expert, followed by Alice, and then Carlos (see Table I). However, when considering

EBD^F, we see that Alice, Carlos, and Bob all have the same knowledge in the project, as counting the number of files modified above the mean by all of them amounts to one in Table II. However, when using EBD^M, we notice that Carlos has the highest expertise, as he is the expert in two out of three files, while the others are expert in only one out of three files, as shown in Table V. Therefore, a deeper analysis at a finer-grain could unveil which developers have breadth of expertise over a large number of methods in the file, and, consequently, should be indicated as an expert of the file. Recursively, developers that have breadth of expertise over a large number of files in the project should be indicated as an expert of the project.

B. Time Matters

As a project evolves, it is possible that developers take on different roles or move to different parts of a project. Typical archival analyses for expertise identification do not take project evolution into consideration [5], [6]. As a result of this, developers who had made frequent changes in the past, but are no longer active and are therefore uninformed of the current project structure, may still be recommended as experts.

We use the notion of a tridimensional matrix, which represents the whole project history and is composed of multiple slices, to consider evolution. Each slice represents a snapshot of the history at a certain point in time. A question that then arises is how should we create a slice to depict the history. On the one hand, humans tend to discretize and think of expertise in terms of time; therefore, we can use time intervals (weeks, months, etc.) to discretize experience. However, on the other hand, the project structure evolves as a sequence of commits (i.e., if no commit is performed in weeks or months, no evolution will be perceived). We reconcile these two factors by computing a slice per unit of time (e.g., one slice per month), but in terms of a sliding window that comprises a set of commits performed before each slice. We define the size of the sliding window as presented in Equation 2.

$$slide\ window\ size = Max(MAM, MLC) * M \qquad (2)$$

Where *MAM* represents the number of commits in the most active month of the project history; *MLC* represents a minimum limit of commits per window, independently of how active a project might be; and M represents a multiplier to allow users to experiment with different window sizes. We identify the number of commits that were performed in the most active month (MAM) of the project and use that as the default size of the sliding window over which we collect commits. As previously explained, we did not simply choose a month as the window size since in open source projects the amount of activity in a given month fluctuates and we wanted to use a constant window size across our calculations. This implies that when we create slices for months that have less activity than the most active-month, they will involve commits from the previous month(s). This is in fact desirable, since having the window overlap across slices smoothens out sharp fluctuations and equally represents the effects of changes over expertise. However, since it is possible that a (small) project might not have a month with enough activity to create an appropriate window size, we use a floor for the minimum number of commits (MNC) that are used to create a slice.

In our illustrative example, performing expertise analysis over such slices might show that Alice has had the highest expertise in *FileC.java* (see layer 1 in Fig. 1), but as we visit subsequent slices we see that the expertise shifts, with Carlos being the expert in the last window (layer 3 in the Fig. 1) of the project. Such a time-based analysis can show when an expertise handover occurs in a project (e.g., between periods 2 and 3 Alice makes much fewer edits to the *FileC.java* and Carlos assumes development for that file). Such analysis can, therefore, identify points when developers started acquiring expertise, how long it took them to gain expertise to the extent of the previous expert, and for how long a developer's expertise is valid (as software changes, past expertise naturally loses strength). Therefore, our approach allows more nuanced investigation of expertise.

FIG. 1. [DEVELOPER|FILE|TIME] BLOCK WITH LAYERS IN THE BACK DENOTING RECENCY .

C. Speed Matters

Processing the aforementioned matrices for large projects, with tens of thousands of artifacts, hundreds of developers, and many years of duration, would require parallel processing instead of a traditional CPU-based architecture. For this reason, we adopted Dominoes [8] to process the matrices for expertise analysis. Due to its GPU-based solution, Dominoes can process large amount of data efficiently.

Dominoes first mines Git (version control repository) to extract the basic relationships among project elements and store them into a relational database (SQLite). It extracts the author and code churn for each commit and then uses the abstract syntax trees (AST) to identify the methods that these changed lines of code belong to. It also extracts the composition structure (packages / files / classes / methods) from the AST. New commits performed on a repository are added and processed incrementally as Dominoes stores the time of the last commit that was processed. After importing a repository, Dominoes represents such relationships as binary matrices called basic building blocks: [developer|commit] (DC), [commit|method] (CM), [class|method] (ClM), [file|class] (FCl), and [package|class] (PCl). Dominoes also allows several operations over data, such as linear transformations (e.g., multiplication and transposition of matrices) and statistics transformations (e.g., Z-score). This way, basic building blocks can be further combined to yield derived blocks that allow exploration of derived project relationships (e.g., [developer|method] = [developer|commit] x [commit|methods]).

We extended Dominoes in this work to include a third dimension to represent time (e.g., [developer|method|time] (DMT)). Its construction is based on the desired relationship, granularity, and timeframe. It basically runs multiple construc-

tions of a given bi-dimensional block (e.g. [developer|method]) for a timeframe (e.g., month), saving them as layers in the 3D block. The timeframe allows us to group data into layers to enable tracking of a project evolution or allow a user to perform an analysis over a unit of time that they care about. The layers in the tridimensional matrix can be data that is grouped by month, week, day, or even a specific number of commits.

In order to allow efficient computation, Dominoes tailored the aforementioned data transformations into a Single Instruction Multiple Thread (SIMT) architecture, making it possible to process the data in GPU devices. Therefore, when a matrix manipulation is needed, Dominoes forks its execution by triggering the respective asynchronous GPU code (called *kernel)* according to the desired operation.

A GPU kernel is implemented in CUDA, a proprietary Nvidia programming library based on C, and is targeted to be executed only over GPUs. Essentially, a CUDA kernel is a function that generates thousands of parallel threads at the GPU device. While all these threads must work over the same code, they operate at different parts of the data. Since modern GPUs can have thousands of cores, when the data is correctly distributed, it is possible to achieve speed-ups of two or even three orders of magnitude when compared with traditional multi-core CPUs [14], depending on the nature of the problem. Modeling the data structure as matrices allows optimal parallelization, especially in the case of operations that have only local data dependencies, as is in our case.

Linear transformations are defined primarily for bi-dimensional building blocks (3D blocks are iteratively constructed per layer). Due to data independence, matrix operations such as transposition and addition are performed in the GPU by $N \; x \; M$ parallel threads, where N and M represents the block dimension in rows and columns, respectively. This independence among rows and columns allows an acceleration directly proportional to the number of available arithmetic cores, when correctly dealing with cached data [15]. Since GPUs may have up to 2000 cores in a single card, it is possible to achieve speed ups of more than 200 times when compared with multi-core CPU architectures [16] for this kind of operation. Additionally, the Z-score operation is defined in GPU for both bi- and tri-dimensional building blocks. Calculating the Z-score requires calculating the mean and standard deviation for each artifact and all commits in a time frame, which can be parallelized reaching $O(\log n)$ complexity [17]. Summing up values in GPU is known as prefix-sum and can be modeled as a parallel reduction problem. The Z-score operation in Dominoes is performed by using Thrust [18], an open source parallel algorithm library.

III. Evaluation

In this section, we present an evaluation of our approach by focusing on the following aspects: (1) fine-grained analysis, (2) analysis based on timeframes, and (3) the use of GPU processing. We used the open source project Apache Derby as our evaluation subject. All analyses were performed over the data extracted from the Derby repository between Aug 2004 and Jan 2014, which comprises **7,573** commits, **36** unique developers, **34,335** file changes, and **305,551** method changes. Apache Derby 10.11.1.1 comprises a total of 2,864 java files with an average of 398 lines of code each. Additionally, we observed that the author of a change is always the committer in the Derby repository, therefore, we only use commit information when determining expertise.

We performed three experiments. The first two experiments analyze how granularity and time may affect the identification of expertise. The third experiment evaluates the scalability of our approach in processing large projects.

A. Granularity

We contrast analyses using fine-grain versus coarse-grain data when identifying expertise in the Derby project by: (1) computing ED and EBD metrics for each file and (2) comparing the experts for each file based on these metrics. ED and EBD metrics diverged in **977** files, amounting to a **28%** difference in expertise calculations. As previously discussed, this difference is a result of how ED is calculated: summing up the number of times a developer edited a file. Because of this, even if a developer has worked on only very specific parts of the file, she is considered an expert for the entire file. This is how expertise is normally identified [5], [10], [11] by current approaches, potentially leading to imprecision in the recommendations.

To demonstrate the difference when we calculate expertise at the coarse-grain (ED) versus at the fine-grain (EBD), let us consider the file "*CreateAliasConstantAction.java*" in the Derby project. This file comprises only one class with four methods: (1) CreateAliasConstantAction(), (2) executeConstantAction(), (3) toString(), and (4) vetRoutine(). Table VI presents the expertise (ED) of two developers who edited this file the most. The mean (1.88) and standard deviation (1.91) were computed out of 17 commits made by 9 developers who edited the file in total.

On the other hand, **Table VII** presents the expertise breadth (EBD) of the same two developers from Table VI as well as the number of commits, mean, and standard deviation in order to highlight the difference between ED and EBD. It is important to note that ED basically represents the number of commits and Z-score of each developer over the file. For instance, 7 commits are equivalent to 2.67 standard deviations above the mean and 3 commits are equivalent of 0.58 standard deviations above the mean.

When we calculate expertise based on the number of commits that each developer made to this file, we see that *djd* has a higher expertise (ED) when compared to *rhillegas*. However, when using EBD we find that *rhillegas* has a higher expertise breadth in the file. The numbered columns show the absolute and Z-score (in parentheses) commits for each method. The last column shows the absolute and Z-score (in parentheses) EBD values for the whole file. As previously discussed, the absolute EBD at file level is the number of methods each developer has modified above the mean (above zero Z-score). In this case, it is possible to see that *djd* has modified just one method above the mean (CreateAliasConstantAction()), while *rhillegas* has modified all of the methods above the mean. Consequently, we can consider that *rhillegas* (EBD value of 4)

has a wider knowledge over this file than *djd* (EBD value of 1).

TABLE VI. ABSOLUTE AND STANDARD SCORE EXPERTISE OF A DEVELOPER (ED) FOR CREATEALIASCONSTANTACTION.JAVA

Developer	*Absolute ED*	*Z-score ED*
djd	7	2.67
rhillegas	3	0.58

TABLE VII. ABSOLUTE AND STANDARD SCORE EXPERTISE BREADTH OF A DEVELOPER (EBD) FOR CREATEALIASCONSTANTACTION.JAVA

Developer	Method 1	Method 2	Method 3	Method 4	*EBD*
djd	1 (-0.57)	2 (0.30)	1 (-0.57)	0 (-0.16)	1 (-1.00)
rhillegas	2 (1.73)	3 (1.50)	2 (1.73)	1 (1.00)	4 (1.00)
Total Commits	5	7	5	1	-
Mean / St. Dev.	1.25 / 0.43	1.75 / 0.89	1.25 / 0.43	1.0 / -	-

It is important to note that, because we are using a fine-grained approach for calculating EBD, we are able to distinguish between modifications that change the method body, as compared to modifications that do not affect methods (e.g., inserting comments or import statements). For instance, in our example we find that *djd* has committed to the file seven times. However, of these only four changes affected methods. On the other hand, *rhillegas* committed the file three times, but each commit changed more than one method (e.g., Method 2 was modified by all three commits). Therefore, when analyzing commits per file, *djd* would be considered an expert, whereas using the fine-grained EBD view, we see that *rhillegas* has a broader expertise. In fact, *rhillegas* has changed each individual method more times than *djd* (i.e., *rhillegas* dominates *djd* in all methods).

Next, we calculate expertise for the file, *EmbedConnection.java*, since it is a large file (comprises 135 methods) and has been edited extensively. Table VIII shows the difference when comparing ED and EBD for this file. From this table, we see that *djd* has the most edits to the file (ED). However, he only touched 18 methods (EBD^M), leading to a low Z-score. In contrast, *kristwaa* committed to this file only 8 times (ED), but her modifications touched 22 methods (EBD), leading to a higher Z-score when compared to *djd*. We find that *rhillegas* has the highest expertise since he edited 59 methods.

Finally, *bpendleton* appears in Table VIII with two commits in the ED list. However, one of his commit was made because of import modifications (not considered when calculating EBD), and the other commit touched just one method. Therefore, he does not appear in the EBD list. In contrast, *coar*, who has just one commit (and 15th position in the ED list), has modifications to 56 methods (2nd position in the EBD list) since he was responsible for the initial code creation.

The same approach can be applied at the project level to identify expertise when considering the entire project history. There are three ways in which we can calculate this, as discussed in Section **Error! Reference source not found.**. First, we calculate the total number of commits performed by each developer in the project (*ED* column) as shown in Table IX. That is, we count the number of commits that has been made in the project by a developer. This metric follows the intuition that the more commits a developer has made to a project, the more knowledge she has about the project.

Second, we calculate EBD for the project by aggregating changes at the file level (i.e., we give expertise credit to a developer for each file, in which she has Z-score of commits above zero for that file). We call this EBD^F. The intuition here is that to be considered an expert, a developer must have changed a file more than average, and that information is aggregated for the project.

TABLE VIII. TOP EXPERT DEVELOPERS AT FILE EMBEDCONNECTION.JAVA BY ED AND EBD^M

Dev.	*ED*	*Z-score*
djd	21	2.28
kahatlen	19	1.98
rhillegas	16	1.53
oysteing	14	1.23
dag	10	0.63
kristwaa	8	0.33
kmarsden	6	0.03
abrown	2	-0.56
bpendleton	2	-0.56
lilywei	2	-0.56
bandaram	2	-0.56
bernt	1	-0.71
davidvc	1	-0.71
dyre	1	-0.71
coar	1	-0.71
mamta	1	-0.71
bakksjo	1	-0.71
suresht	1	-0.71
tmnk	1	-0.71

Dev.	*EBD^M (Method Level)*	
rhillegas	59	2.04
coar	56	1.89
kahatlen	50	1.59
kristwaa	22	0.21
dag	22	0.01
djd	18	0.01
oysteing	18	0.10
kmarsden	8	-0.47
bernt	4	-0.67
bandaram	2	-0.77
lilywei	2	-0.77
tmnk	1	-0.82
suresht	1	-0.82
abrown	1	-0.82
mamta	1	-0.82
dyre	1	-0.82

Finally, we calculate EBD for the project by aggregating changes from the method level (EBD^M). That is, we give expertise credit to a developer for a file if she has a Z-score above zero when considering changes to methods of that file. The intuition here is that a developer is considered an expert in a file if she has breadth of knowledge in that file (Zscore > 0), and this is recursively computed for the project.

When we compute the top 10 experts in the project, we see that *kahatlen* appears as the highest expert in the project when considering the ED measurement. However, in EBD^F and EBD^M he moves to the 3rd position. Conversely, *djd* assumes the 1st position for both EBD^F and EBD^M measures, which means that *djd* has more breadth in knowledge across the whole project than *kahatlen*. This clearly shows that calculating expertise by simply computing the number of commits performed by a developer can yield very different results. Moreover, *coar*, who appears in the 5th and 2nd positions in EBD^F and EBD^M, respectively, does not even appear in the ED list. The same situation occurs with *dag*, who moves from the 8th position according to both ED and EBD^F to the 5th position when considering EBD^M. Also note that *bandaram* (colored in green) in the EBD^M list does not appear in either ED or EBD^F lists. On the other hand, two other developers in EDB^F list (*bakksjo* and *davidvc*, colored in red) do not appear in the EBD^M list. This shows that many developers make edits to only some portions of the file and may not have expertise over the majority of the methods in a file.

Another interesting case occurs when we consider *bakksjo*, who is at the 4th position in the EBD^F list with a Z-score of 1.52. When considering EBD^M, he falls down to 26th position in this list (not shown in Table IX), presenting a value of 8 and a Z-score of -0.61. This happens because *bakksjo* has modified

over 1300 files. However, most of his changes introduced comments and copyright modifications, without any functionality changes in the code. This is another example of how simply calculating edits to a file may not be the right approach to identifying expertise on that file. These results show how fine-grained analyses at the method level can provide a more nuanced view of expertise. Consequently, another advantage of analyzing edits at the method level is the ability to filter out non-code related changes. Previous research has used commit size as a mechanism to filter out copyright and other non-code related changes [7], [19]. Our results show that that analyzing edits at the method level provides a more accurate means of filtering out these ancillary commits.

TABLE IX. TOP 10 EXPERT DEVELOPERS BY ED, EBD^F AND EBD^M CONSIDERING ABSOLUTE AND Z-SCORE VALUES

Dev.	*ED*
kahatlen	1393
djd	1190
rhillegas	990
kmarsden	650
kristwaa	640
fuzzylogic	412
myrnavl	385
dag	**331**
mikem	**284**
mamta	282

Dev.	*EBD^F (File Level)*	
djd	1846	2.61
rhillegas	1780	2.49
kahatlen	1556	2.08
bakksjo	1252	1.52
coar	**1210**	**1.45**
kristwaa	1045	1.15
fuzzylogic	893	0.87
dag	**879**	**0.84**
davidvc	866	0.82
bpendleton	823	0.74

Dev.	*EBD^M (Method Level)*	
djd	1533	3.42
coar	**1244**	**2.66**
kahatlen	1047	2.13
rhillegas	960	1.90
dag	**631**	**1.03**
kmarsden	467	0.60
kristwaa	403	0.43
bandaram	361	0.32
fuzzylogic	322	0.21
bpendleton	1245	0.01

B. Time

Another important aspect to consider when analyzing expertise is time. As software and people change over time, it is expected that expertise also evolves. The Apache Derby project, which is a long living and active project with a 10-year history, is not an exception. We analyze expertise evolution through two studies, which analyze changes in expertise: (1) in one particular file, and (2) the entire project. For both analyses we group commits by using Equation (2).

We select the file *EmbedConnection.java* to measure expertise evolution when considering a single file, because it was the third most committed file in the project and was "touched" by a large number of developers. When calculating the size of the sliding window for the analysis (see Equation 2), MAM equals to 11, as this was the number of commits that occurred in the most active month, which was Feb 2008. MNC equals to 10, as we believe this provides a sufficiently large slice of data over which to perform analysis. Note that our choice of MNC does not influence the results, as it is lower than MAM. We did not change the M factor; therefore, our sliding window size is 11.

We derive the [Developer|Method] blocks for each layer, based on the aforementioned sliding window and stack them to create a [Developer|Method|Time] 3D block for the *EmbedConnection.java*. We then compute a time-based EBD for it.

Fig. 2 shows the graph of EBD evolution of developers who have edited *EmbedConnection.java*. We observe that *djd* was the most active developer for almost half a year. Then, onwards *kahatlen* assumed expertise for almost a year, followed by *rhillegas*, both relinquishing their expertise at the same time. The graph shows that *rhillegas*, after almost three years since the beginning, starts to become an expert again in this file. A few months after this *oysteing's* expertise starts to decline. At the end, *rhillegas* can be considered the expert in this file. Note that if we were to determine an expert for the file by using ED (and over the entire project) *djd* would be considered the expert (Table VIII). On the other hand, when we consider the evolution of expertise (using EBD), we find that *rhillegas* has the most expertise breadth at the end.

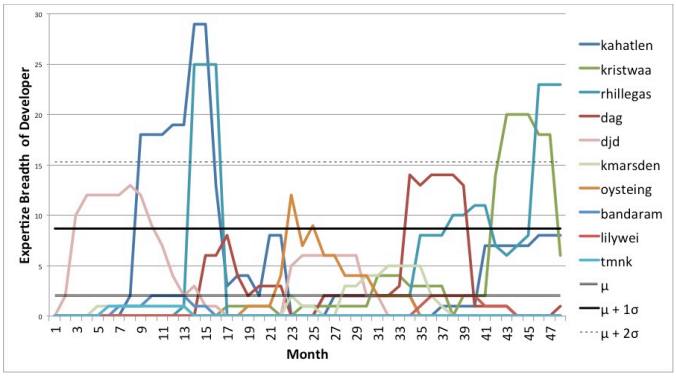

FIG. 2. DEVELOPER BREADTH EXPERTISE FOR FILEEMBEDCONNECTION.JAVA.

For the second study (temporal analysis of expertise across the project as a whole), when we consider the sliding window for analysis we group commits by setting MAM to 270 (Equation 2). This was the maximum number of commits that occurred in the most active month (Aug 2006). We set MNC to 100, as we believe this provides a sufficiently large slice of data over which to perform analysis. Again, our choice of MNC does not influence the results, as it is lower than MAM. Also, the M factor was not changed, leaving the sliding window size to 270.

We then compute EBD for the entire project by considering each time slice. Here we show the results of our analysis of the top five developers in terms of EBD in Table IX.

Fig. 3 shows evolution of EBD of these developers. In this graph, we find that *djd* was the main developer in the beginning of the project, together with *coar*. Almost a year and a half later, *rhillegas* and *kahatlen* start to contribute more extensively to the project, increasing their expertise. Four years after the start of the project, *djd* leaves the project and *rhillegas* takes over the development. In the fifth year of the project *rhillegas* has his first month with EBD higher than two standard deviations above the mean (higher dashed line). While *rhillegas* still remains very active, after six years *kahatlen* seems to supersede *rhillegas* in terms of EBD. This clearly contrasts with the results presented in Table IX. When we do not consider evolution, *djd* would be considered the main expert.

Finally, *dag* appears as the leading expert in Derby's recent years (superseding *kahatlen*), but had less activity in the earlier years. His expertise hardly even crossed one standard deviation above mean (lower solid single line) during the project. Nevertheless, he was an active developer. However, when analyzing over the entire project history, *dag* (see Table IX ED column) is classified as the 8th expert in the project.

C. Performance

Our approach has been designed to enable online exploratory analysis over large-scale software engineering data. It makes efficiency of computation an underlying requirement. In this section, we provide some benchmarks contrasting our approach using Dominoes with an equivalent tool in CPU. This tool was specifically developed to allow comparison of optimal CPU implementations with our GPU implementation.

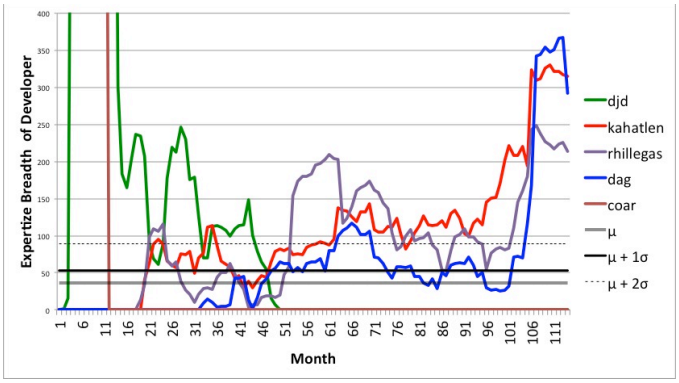

FIG. 3. DEVELOPER BREADTH EXPERTISE FOR THE WHOLE PROJECT.

In order to make a fair comparison, all linear transformations and matrix operations in CPU are made in OpenBLAS[2], an open source implementation of BLAS (Basic Linear Algebra Subprograms) API with many handcrafted optimizations for specific processor types. OpenBLAS is able to decompose a BLAS operation into smaller "kernel routines" and is thus able to use all available CPU cores during its processing, thereby making it a fair comparison. We experimented both tools with different matrix sizes, providing evidences regarding scalability. We use the **Speed up** metric, defined in Equation (3) to measure how fast GPU is in relation to CPU.

$$Speed\ Up = \frac{Time^{CPU}}{Time^{GPU}} \quad (3)$$

In this experiment, we used a PC equipped with an Intel Core 2 Quad Q6600 using 8 GB of RAM and an nVidia GeForce GTX 580 model, which has 16 Stream Multiprocessor with 512 Stream Processor each.

The initial (full) loading of all commits into the database required 15 minutes in order to create the AST, while the parser to identify all changed methods took about 208 minutes. However, we reiterate that this process occurs only once, as future commits are processed in an incremental way. After that, we measure the total time spent to process the 3D building block introduced in Section II.B at the coarse level ([Developer|File|Time]). This block comprises 114 layers of a 36 x 3400 ([Developer|File]) matrix (a total of 13,953,600 elements). Building this block from Dominoes' database takes 2.38 seconds. On the other hand, composing a 3D building block at fine grain level ([Developer|Method|Time]) takes about 189.96 seconds, comprising of 114 layers of 36 x 43,788 matrices ([Developer| Method]) each. Building block generation is always done in CPU, as it is a query over the database.

[2] OpenBLAS: http://www.openblas.net

Processing the Z-score requires calculating the mean and standard deviation of each layer and then computing the Z-score itself. Table X shows the times taken to calculate EBD. In the first column ("EBD when Considering Files"), we show the time necessary for creating a 3D block for the entire project when considering files as the unit of analysis. The total time taken to process all layers ("Total" column) is 303.42 versus 19.59 seconds when comparing between CPU and GPU, respectively. Therefore, we have a speed up of 15.48 times in relation to CPU. In the second column ("EBD when Considering Methods"), we also calculate EBD for the entire project, but this time considering methods as the unit of analysis. The total time taken to process all layers ("Total" column) is 1,998.31 and 212.01 for CPU and GPU, respectively, achieving a speedup of 9.42 in relation to CPU.

The results show that a large difference in performance can be achieved when we perform the same analysis in GPU as compared to using CPU, even when the latter employs optimal algorithms provided by OpenBLAS. It is important to note that we used a conventional and affordable GPU card when running the experiments. Replacing this card with a more powerful card, such as nVidia Tesla K40, could easily further boost Dominoes performance. The nVidia Tesla K40, the most powerful GPU at the time of writing of this paper, provides at least 20 times more computational power.

TABLE X. PROCESSING TIME (IN SECONDS) FOR CALCULATING 3D BUILDING BLOCKS FOR EBD IN THE PROJECT

	EBD when Considering Files			***EBD when Considering Methods***		
	Mean & SD	*Z-Score*	*Total*	*Mean & SD*	*Z-Score*	*Total*
CPU	2.19	301.23	303.42	424.71	1,573.60	1,998.31
GPU	0.10	19.49	19.59	8.55	203.46	212.01
Speed Up	21.90	15.45	15.48	49.67	7.73	9.42

IV. THREATS TO VALIDITY

The divergence of expertise from expertise breadth that we observed is largely based on the Apache Derby project. We selected Apache Derby because it has a large, active contributor base. It is possible that other projects might not have as much as deviation in expertise breadth. We need further studies of other projects to observe whether such deviation between expertise and expertise breadth holds true. A central construct in our analysis is that if someone edits a method, we assume that the developer has a certain degree of knowledge of that method. However, since other approaches also use number of edits as a proxy for knowledge, this is not a major threat. Further, our analysis filters out non-source changes to the method (or file).

Another threat in our study is that we use Z-scores to identify expertise, which assumes a normal distribution. Project data might not be normal. However, since we use Z-scores to identify developers who have made edits more than the average number of times, non-normality of our data should not be a problem. In the worst case, the Z-score provides us a ranking (even if some developers have negative scores) with a perception of the distance among data points in the sample.

V. Related Work

This section present related works divided into three groups: (1) research that deal with expertise identification, (2) infrastructures for repository analysis and (3) infrastructure focused on speeding up data analysis.

McDonald and Ackerman [7] designed Expertise Recommender (ER), which is based on two heuristics for recommending developers for specific tasks: tech support and change history. The tech support heuristic uses an issue database to search for similar past task and recommends developers involved in the previous tasks. The change history heuristic states that the last person that changed an artifact is a good candidate for changing it again. Unfortunately, the latter heuristic puts an extremely high weight on recent changes and ignores the past. Our approach uses the entire history of a repository, but segregate this history into timeframes that allow the perception of how expertise fluctuates over time.

Gîrba et al. [19] defined an "Ownership Map" visualization to understand when and how different developers interacted and in which parts of the code, as well as define who a file belonged to and for how long. In their approach, lines of code added/removed by each developer are counted to determine ownership. A developer who owns (made the latest edit) the most number of lines in the file is considered the owner of the file. It does not take into account the syntactic structures of the file (such as classes or methods) and is influenced by non-functional changes like comments. Our approach, in contrast, can detect changes at different granularities as well as ignores non-functional changes.

Anvik et al. [4] proposed an approach to recommend developers for a specific issue by using machine learning techniques exclusively over an issue database. This way, the information provided does not take into account artifacts' modifications in order to suggest a developer who is the most appropriate to change specific artifacts. Our focus is different, as we identify expertise over artifacts (e.g., files, project).

Posnett et al. [20] considers both artifact and developers perspectives in order to extract focus and ownership for computing a unified score: DAF (developer attention focus), which measures how focused is a developer during his task (i.e., their work is spread among many artifacts or more focused). In contrast, EBD measures the breadth of expertise of a developer in specific artifacts (or the project).

Schuler and Zimmermann [21] proposed an approach for measuring expertise by the frequency in which a developer uses a method. While this information can help identify a person who knows how to use a method, it might not help in identifying the developer who knows that method the best, that is, the person who is the most appropriate to edit that method. Instead, our approach uses modifications inside a method for proposing experts in a file.

Additionally, infrastructure to facilitate automated expertise identification through repository exploration exists. For example, Minto and Murphy [5] introduced the Emergent Expertise Locator (EEL). EEL is based on the framework by Cataldo et al. [6] for matrix manipulation, thus requiring massive linear operations to be performed depending on the size of the project. To avoid this problem, EEL imposes a constraint over matrix size, allowing only matrices up to 1,000 × 1,000 to be used. Besides that, EEL uses coarse-granularity (i.e., files) to recommend experts. However, the problem of personnel allocation becomes harder for large projects, with much more than 1,000 files. Moreover, assuming files as atomic units may lead to inadequate recommendations, as changes in very specific parts of the file or broader changes in multiple parts are considered equivalent. Our proposed approach, on the other hand, works at a fine grain (i.e., methods) to differentiate specific changes from broader changes. This leads to large matrices that need to be processed, which can be performed interactively because of the underlying GPU architecture of Dominoes.

Kagdi et al. [3] proposed a system for assisting in the tasks of allocating developers for changing a given file. It considers three metrics to compose a ranked list of recommended developers: contribution, activity, and recency of changes. The contribution metric indicates the number of commits each developer performed over the file. The activity metric indicates the number of days the developer has committed at least once in the project. The recency metric indicates the date of the last commit of each developer. Similar to the previously discussed approaches, this approach works at coarse grain (i.e., files) and is not designed to support interactive exploratory analysis.

Research has produced several infrastructures designed for repository analysis. For example, Kenyon [22], is a system that integrates information originating from various repositories into a single database for future analysis. Our approach takes a step further and allows expertise analysis using data from the repository at interactive speeds. Evolizer by Gall et al. [23] allows analysis of software archives. It analyzes AST level changes to identify different types of changes and modification patterns. Our approach also uses AST to identify relevant changes, but with the focus on expertise identification. CodeBook by Begel et al. [24], creates a network that connect developers and artifacts by mining version control change logs, emails, and other artifacts in a software repository. The underlying graph can then be used by applications to answer different analysis questions (e.g., WhoseIsThat [25] identifies artifacts edited by a developer). Our approach uses matrices to define software relationships and allows for incremental update of data, which is not the case for CodeBook.

Finally, there exists research that aims to speed up repository analysis. Boa [26] provides an infrastructure for analyzing large scale software repositories on a cluster. Although its performance may be comparable to Dominoes, setting up a CPU cluster is not a trivial task, making it more applicable for researchers in the university than for developers in the industry. Jean-Rémy et al. [27] developed the Harmony platform, an unified model that extracts and analyzes data at a coarse grain from different version control systems. After extracting the data to a database, the user is responsible for dealing with pieces of data to extract the desired information. These tools require the user to write a functional script to define what the platform needs to process per analysis type. This can be a constraint, since end users will need to program their analysis and write a script for each of their queries.

VI. Conclusion and Future Work

Performing exploratory data analysis in software repositories has computational challenges. Here we show how our approach can analyze fine-grained data, such as edits to classes or methods to identify expertise in a project. Our results show that when we consider expertise by only recognizing edits at the file-level, we get a **28%** deviation as compared to when we analyze expertise based on the breadth of developers' knowledge. Moreover, since we calculate changes at a fine-grained level, we are able to filter out those changes that do not affect the method body, thereby, being more precise in expertise identification. In order to validate our results, we plan to interview Derby managers and collect their interpretation about expertise breadth versus depth. Additionally, we plan to conduct new exploratory data analysis in other projects.

We also introduce the concept of tri-dimensional matrices of relationships across software project elements over time. These matrices allow temporal analysis of relationships. When identifying expertise in Apache Derby, we found that temporal analysis shows the fluctuations in developer expertise for a single file, as well as for the whole project. It also shows when a developer stopped being active and "handed over" the expertise to another. Had we considered the entire history of the Derby project, the analysis would incorrectly recommend a developer who was no longer active in the project.

We know that each method inside a class might not have the same importance. For instance, *get/set* methods have a low complexity to implement or maintain, when compared to other functional methods (e.g., executeConstantAction*)*. In the future, we plan to use a weighting mechanism for methods according to their complexity, which can then help us in identifying experts not only on the breadth of their knowledge, but also on the complexity of the changes that they have made. We also plan to perform our analyses on other projects to generalize the feasibility of our approach, as well as for showcasing differences when calculating expertise by considering low level changes and breadth of changes. In addition, we plan to conduct a case study with industry managers to identify the usability of our analyses and the presentation of the results.

Finally, using Z-score for data with a non-normal distribution can be a problem depending on the spread of the data. To mitigate this problem, we plan to investigate the adoption of the modified Z-score, which uses median and MAD (Median Absolute Deviations) [28] instead of mean and standard deviation.

References

[1] J. D. Herbsleb and R. E. Grinter, "Splitting the Organization and Integrating the Code: Conway's Law Revisited" in 21st ICSE, 1999, 85–95.

[2] A. Meneely and L. Williams, "Socio-technical Developer Networks: Should We Trust Our Measurements?" in 33rd ICSE, 2011, 281–290.

[3] H. Kagdi, M. Hammad, and J. I. Maletic, "Who can help me with this source code change?," in IEEE ICSM, 2008, pp. 157–166.

[4] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix This Bug?," in 28th ICSE, 2006, pp. 361–370.

[5] S. Minto and G. C. Murphy, "Recommending Emergent Teams," in Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07, 2007, pp. 5–5.

[6] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the Design of collaboration and awareness tools," in the 2006 20th anniversary conference on Computer supported cooperative work, 2006, 353–362.

[7] D. W. McDonald and M. S. Ackerman, "Expertise Recommender: A Flexible Recommendation System and Architecture," in the 2000 ACM Conference on CSCW, 2000, pp. 231–240.

[8] J. R. da Silva Junior, L. Murta, E. Clua, and A. Sarma, "Exploratory Data Analysis of Software Repositories via GPU Processing" in 26th SEKE, 2014.

[9] S. Rajasekaran, L. Fiondella, M. Ahmed, and R. A. Ammar, Multicore Computing: Algorithms, Architectures, and Applications. New York, NY: CRC Press, 2013.

[10] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity," in Proceedings of the Second ACM-IEEE ESEM, 2008, pp. 2–11.

[11] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in the 31st ICSE, 2009, pp. 23–33.

[12] J. Anvik and G. C. Murphy, "Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions," ACM Trans Softw Eng Methodol, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011.

[13] S. R. Carroll and D. J. Carroll, Statistics made simple for school leaders data-driven decision making. Lanham, Md.: Scarecrow Press, 2002.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General-purpose Applications on Graphics Processors Using CUDA," J Parallel Distrib Comput, vol. 68, no. 10, pp. 1370–1380, Oct. 2008.

[15] N. Corporation, NVIDIA CUDA C Programming Guide. 2014.

[16] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication," in The ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2004, pp. 133–137.

[17] H. Nguyen and NVIDIA Corporation, GPU gems 3. Upper Saddle River, NJ: Addison-Wesley, 2008.

[18] J. Hoberock and N. Bell, Thrust: A Parallel Template Library. 2010.

[19] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How Developers Drive Software Evolution," in PSE, 2005, vol. 0, pp. 113–122.

[20] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual Ecological Measures of Focus in Software Development," in The 2013 International Conference on Software Engineering, 2013, pp. 452–461.

[21] D. Schuler and T. Zimmermann, "Mining Usage Expertise from Version Archives," in The 2008 International Working Conference on Mining Software Repositories, 2008, pp. 121–124.

[22] J. Bevan, E. J. Whitehead Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution Research with Kenyon," in The 10th European Software Engineering Conference, 2005, pp. 177–186.

[23] H. C. Gall, B. Fluri, and M. Pinzger, "Change Analysis with Evolizer and ChangeDistiller," IEEE Softw, vol. 26, no. 1, pp. 26–33, Jan. 2009.

[24] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and Exploiting Relationships in Software Repositories," in The 32Nd ACM/IEEE ICSE - Volume 1, 2010, pp. 125–134.

[25] A. Begel, K. Y. Phang, and T. Zimmermann, "WhoselsThat: Finding Software Engineers with Codebook," in The 18th ACM SIGSOFT FSE 2010, pp. 381–382.

[26] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories," in the 2013 ICSE, 2013, pp. 422–431.

[27] J.-R. Falleri, C. Teyton, M. Foucault, M. Palyart, F. Morandat, and X. Blanc, "The Harmony Platform," CoRR, vol. abs/1309.0456, 2013.

[28] D. C. Howell, "Median Absolute Deviation," in Encyclopedia of Statistics in Behavioral Science, John Wiley & Sons, Ltd, 2005.