I. INTRODUCTION

Software repository mining techniques can provide insights about software systems and their development processes through the use of metrics that aim to capture a construct of interest. However, linking development history metrics with high-level constructs is fraught with threats to validity. We conducted a case study in which we performed a critical review of the underlying artifacts used to compute a metric of knowledge at risk in software projects, proposed in prior work. The case study revealed eight major threats to validity that have the potential to generalize to other software process metrics derived from repository data. In addition to a detailed description of each threat, we contribute a questionnaire to facilitate their assessment in past and future studies.

Index Terms—Mining software repositories, Software metrics, Knowledge loss, Threats to validity.

II. KNOWLEDGE LOSS CASE STUDY DATA

Using the computation of Knowledge at Risk as a case study, we set out to investigate, in depth, the threats to validity associated with metrics that summarize typical software repository data: commits to files by contributors. Using as a starting point our replication [6] of Rigby et al.’s study [4], we retrieved and analyzed every file considered abandoned to investigate the properties of the files, including size, file type, and amount of comments. We conducted our analysis in two phases, termed macro- and micro-analysis. The goal of the macro-analysis (Section III) was to identify threats to validity by considering the entire data set, but along new dimensions that leveraged our extended set of file properties. The goal of the micro-analysis (Section IV) was to delve even deeper into the data, which can only be made tractable by focusing on a subset identified using a systematic procedure.

As a result, we documented eight specific threats to validity associated with the aggregation of software repository data (Section V). Although we identified the threats in the context of a case study of knowledge at risk, our approach supports a straightforward analytic generalization [7] to other metrics based on large-scale data aggregation from software repositories. We thus articulate our results as general threats, and develop them into an evaluation instrument designed to help mitigate these threats in future studies.

The contributions of this paper include a) a critical re-analysis of previously-published file abandonment computations for eight open-source systems; b) a new approach for analyzing software repository data based on a combination of macro- and micro-analysis; and c) a description of eight major threats to validity for software process metrics derived from repository data together with a questionnaire to facilitate their assessment in past and future studies.
possibility that the KaR value may not really be representative of the risk of knowledge loss.

**Basic Concepts for Computing Knowledge at Risk**

The period is an arbitrary time interval of the development history of the project during which contributors can leave the project and thus become leavers [4], [8]. During a period, a project comprises several source files. Each line in each file can be attributed to (or owned by) one contributor. Because ownership can be detected at the granularity of lines, the ownership attribute for a contributor is the number of lines they own. If a line is owned by a leaver, it is said to be abandoned. If a file contains a high proportion of abandoned lines, it is said to be abandoned.

**Extended Data Model**

Figure 1 represents the data model that we instantiate for this study. This model adds two main additional types of information to the basic associations between periods, files, and authors used in the initial study. First, we add file properties, including the file type (represented by its extension, e.g., “.cpp”), the length of the file (in lines), the number of characters of code in the file (codeChars) and the number of characters of comments in the file (docChars). The rationale for considering file properties is that files are not atomic building blocks for software systems.

Second, we add a map between a file and a general concept of module. The rationale for considering modules is that files are rarely independent, and metrics that are based on files can lead to different interpretations whether or not these files are in the same module. For example, the impact of a leaver who owns a few files in many modules may be different from the impact of a leaver who owns many files of a few modules. We use the module-related information for the micro-analysis in Section IV.

We instantiated the data model for all eight projects in our earlier replication study [6]. This work required a combination of straightforward extraction of the available data combined with new extraction procedures for the file properties and modules. Table I, adapted from the original work [6, Table 1], summarizes relevant project characteristics.

**Basic Data Extraction**

We used the replication package to obtain, for each project, the lists of (a) periods studied and their timestamps, (b) leavers for each period, and (c) files with their corresponding ownership information for each period. In this data model, periods are three months in length and files are considered abandoned if 90% of their lines are owned by leavers.

**Extracting File Properties**

To complete the model with file properties, we cloned each project from their Git repository to retrieve all the project’s files for each period. Using the ownership information in the replication package, we compiled the list of all files in a project at the end of each period, and used the `git-show` command to retrieve the content of the file at the required timestamps.

To derive the type of the file, we used the extension of the file. This yielded a set of 20 different file types (e.g., “.cpp”, “.cmake”). To obtain the number of characters of code and comments, respectively, we wrote lightweight parsers. The parsers read in each character in a source file, skip all whitespace characters, and sum up the characters in comments and non-comment regions, respectively. The parsers also skip copyright blocks using a simple heuristic: if the first block comment of a file contains the keyword “copyright”, it is eliminated from further consideration.

**Creating the Module Map**

Because of the large size of some of the systems under investigation, we were also interested in partitioning the analysis in terms of modules. For this purpose, we needed a mapping between files and modules. Ideally, such a mapping would be constructed manually, by an architect of the system. For practical reasons, this strategy was not possible. Additionally, because partitioning files into modules can have an impact on the subsequent investigation, we sought a transparent and replicable process for identifying modules, that would reduce threats of investigator bias. We therefore designed the following procedure, which aims to be conceptually simple, partition the files in a project into a manageable number of modules, have modules of manageable size, and have modules that align with the general structure of the project. The general principle guiding our algorithm is to identify as a module any subdirectory of the project with between 2% and 20% of the total file namespace.
More specifically, for a given project:

1) We aggregate all of the files in each period under study into a single, merged directory tree. We then count the total number of files in this directory tree, which will henceforth be referred to as $total$.

2) We create an initial map that considers all first-level directories to be modules. We compute the cardinality of each such module, and place all of the files that are located directly in the root directory in a generic module called General.

3) We remove the mapping for any module whose cardinality is less than 2% of $total$. The corresponding files then become mapped to the General module.

4) For any module with cardinality of 20% or more of $total$, we create a module mapping for any subdirectory with cardinality of 2% or more of $total$. The remainder of the subdirectories get automatically mapped to the parent module, unless the remaining cardinality in less than 2%, in which case the parent module is removed and the remaining files are mapped to the General module.

The Trinity project was the only case where the procedure was modified, where an sql folder was excluded from the calculation because it contains a large number of source files that are versioned clones of each other.

**Replication Package**


### III. Macro-Analysis of File Abandonment

Empirical work on knowledge at risk concludes that the studied software projects are susceptible to large knowledge losses based on the observation that the probability distributions of file losses tend to have “spikes” that show a non-negligible probability of massive losses [4], [6]. However, it is not clear what these spikes truly indicate. In the first part of this case study, we dissect the probability distributions for knowledge at risk, as an exemplar metric, by reviewing the entire data set that was used to compute the metric.

**File Abandonment Distributions**

The visualization in Figure 2 summarizes the number of abandoned files for all of the studied projects. Each bar represents a time period in the history of the system, in chronological order from left to right. The height of the bar shows the number of files that became abandoned in that period (i.e., files that were abandoned in the period, and were either nonexistent or not abandoned in the previous period). To give a sense of the importance of file abandonment at any point, the dashed line shows 5% of the total number of files in the system at the end of each period. The two shades represent the distinction between implementation and non-implementation files, discussed below.

For each studied project, with the exception of Kodi, we observe that there is at most one period exhibiting large losses on the order of 5% or more of the system’s files.

Fig. 2. File Abandonment per period. Each bar represents a time period in the history of the system, in chronological order from left to right. The height of the bar shows, respectively, the number of implementation (black) vs. non-implementation (gray) files that became abandoned in that period. The dashed line shows 5% of the total number of files in the system (both implementation and non-implementation) at the end of each period.
**Fragility Threat:** The transformation of a small number of rare events into a probability may misrepresent the underlying phenomenon.

In our case, the precise probability is simply an artifact of the number of periods considered.

**Implementation vs. Non-Implementation Files**

After a preliminary manual investigation of abandoned files, we realized that many of these files were unlikely to contain software development knowledge at risk. In particular, we noted that in addition to implementation source code, abandoned files also consisted of build scripts, tests, or object data, or were small stubs or boilerplate code (e.g., exception class declarations, enumerated values, etc.). To investigate this issue, we created a distinction between implementation and non-implementation files. We automatically classify a file as a non-implementation file if:

- It is a data or build file according to its file type;\(^1\)
- It is a test file, as determined by the presence of the string “test” in the file’s path.
- It is a tiny file, which has less than 10 lines or less than 50 characters of non-comment content. These thresholds were selected after a preliminary inspection of the files, to represent a file equivalent to a class stub.

In Figure 2, the black portion of the data bars shows the proportion of abandoned files that were implementation files. Table III adds some precision to the analysis. In the table, Largest Event provides the maximum number of implementation files that were abandoned in one period, and the percentage of the total project it represents. Implementation is the ratio of abandoned files that were implementation files over all instances of file abandonment.

We make three observations about the ratio of abandoned files that are implementation files. First, for some systems, the ratio of implementation meaningfully affects the amount of abandoned files; For both Gitlab and Trinity, considering only implementation files would decrease the loss ratio by close to 50% for the period with the largest amount of abandonment. Second, the ratio of implementation files varies across periods. A stable implementation ratio across periods would support an easy correction of corresponding metrics. However, for most systems, the abandonment history shows periods where almost all abandoned files are implementation files (entire black bars), whereas in other periods the ratios vary, somewhat unpredictably. Third, the ratio of implementation files varies across systems. This observation is best made using the implementation column of Table III. At the lower extreme, we find Trinity, whose lower proportion of implementation files is explained by the reliance of the project on a large number of short SQL scripts. On the other hand, Linux and Kodi have large proportions of implementation files because of the almost complete lack of tests in the projects at the time of analysis.

\(^1\)That is, it has one of the following file extensions: .cob, .object, .x, .am, .in, .classic, .cmake. We only considered the types of files in the data set, so this list is internally exhaustive.

**File Content Threat:** The quantity and complexity of code is not uniformly distributed across files.

### Size Distributions

It is clear that the File Content threat also results from variations in file size. We assessed the impact of this threat for knowledge loss. Figure 3 shows the distribution of file instance sizes (in number of lines), with a distinction between abandoned and owned (i.e., not abandoned) file instances. We use the term file instance to clarify that in our analysis a data point is a file in a given period, so for any file that survives a single period, the simple term file becomes ambiguous. The figure thus represents all file instances across all periods for a given project. Within the figure, we also report the p-value of an unpaired Wilcoxon test that gauges the probability that owned and abandoned file instances are issued from the same size distribution, and the interquartile range (IQR) of the distribution of the sizes of abandoned file instances. We employ a logarithmically scaled Y-axis and use interquartile ranges as a measure of dispersion because the distribution of file instance sizes is skewed, with some systems exhibiting very large file instances.

From the figure, we readily observe that half of the systems have interquartile ranges in the hundreds of lines. Interestingly, we also observe that abandoned file instances are statistically significantly shorter than non-abandoned file instances (the only exception is for Assimp, which is by far the smallest project). However, this appears to be a good example of a statistically significant yet practically insignificant result. First, the effect size is minimal, and second, the result is at serious risk of being an artifact of the bias of large files, an issue that we now discuss in detail.

### Large Files

The distributions in Figure 3 include some unusually large values, such as a 200kLOC file instance in Chromium and a 176kLOC file instance in Trinity. We investigated these unusual data points by manually inspecting every file instance over 20kLOC. Of the 291 file instances that exceed this threshold in the studied projects, only three were abandoned, so clearly very large files are unusual data points.

To facilitate the analysis of large files, we first merged all duplicates (same path in different periods), and obvious clones (same name stem in different directories, identical or

<table>
<thead>
<tr>
<th>System</th>
<th>Largest Event</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apereo</td>
<td>77 (13%)</td>
<td>77%</td>
</tr>
<tr>
<td>Assimp</td>
<td>34 (6%)</td>
<td>80%</td>
</tr>
<tr>
<td>Chromium</td>
<td>479 (2%)</td>
<td>70%</td>
</tr>
<tr>
<td>Gimp</td>
<td>138 (6%)</td>
<td>81%</td>
</tr>
<tr>
<td>Gitlab</td>
<td>15 (3%)</td>
<td>66%</td>
</tr>
<tr>
<td>Kodi</td>
<td>1046 (44%)</td>
<td>93%</td>
</tr>
<tr>
<td>Linux</td>
<td>1837 (7%)</td>
<td>98%</td>
</tr>
<tr>
<td>Trinity</td>
<td>198 (3%)</td>
<td>53%</td>
</tr>
</tbody>
</table>
Fig. 3. Size distribution of file instances collected from all periods. The right box plot shows the size distribution of abandoned file instances. The p-value was computed using an unpaired Wilcoxon test. The interquartile range values for abandoned file instances is also included.

Fig. 4. Comment distribution of file instances. For a project the left box plot shows the comment ratios distribution of owned file instances. The right box plot shows the comment ratio distribution of abandoned file instances. The p-value was computed using a Wilcoxon test. The interquartile range for abandoned file instances is also included.

expressing the likelihood that the distributions are from the same population according to an unpaired Wilcoxon test, and the interquartile range for abandoned file instances.

The data shows considerable amounts of variability both between and within projects. In terms of between-project variability, we first notice major differences in commenting practices, from minimal (Gimp) to extensive (Assimp). Although the fact that different projects have different commenting practices is hardly surprising, the analysis provides a concrete illustration of the difference in expected impact for file abandonment between projects. The same reasoning can be made within a project, given that we observe large interquartile ranges in some projects. For example, in the case of Assimp, half of the comment ratio distribution is between 0.179 and 0.660, which exposes particular file abandonment events to a large amount of uncertainty about the true amount of undocumented knowledge in the file.

As an illustration of the meaning of comment ratios, we consider two abandoned files from Assimp. One file, AiConfigOptions.java, has 662 lines when it becomes abandoned, so it could naively be considered to represent an important amount of knowledge at risk. However, upon closer inspection, we see that it is an enumerated type, where 98% of the lines in the file are comments. At the other end of the spectrum, we have OgreStructs.cpp, a 1.2kLOC implementation file with very few comments (6%).

**File Role Threat:** File type is an imperfect way to detect the role that a file plays in a system. Review by project experts may be necessary to reliably recover file roles.

**Comment Ratio Distributions**

Source files contain a combination of formal and informal information (i.e., code vs. comments), a distinction that is not always considered when aggregating process metrics from software repositories. We investigated this issue by analyzing the comment ratios for each file in our data set. For our case study, the relevance of comment ratios is based on the hypothesis that if a file is commented, the knowledge that it contains has been partially externalized (if not perfectly), and is thus at a lower risk of loss because even if the file is abandoned, part of the knowledge that it captures is explicit.

We define the comment ratio of a file as the number of non-whitespace characters within syntactically-defined comments (excluding the copyright block), over the total number of non-whitespace characters in a file (still excluding the copyright block). Figure 4 presents the results, also with the p-value.
Identifying File Abandonment Events

We sought to sample large file abandonment events which, according to the hypothesis of Rigby et al. [4], could threaten the integrity of a project. We were also interested in sampling from different modules and different periods. To keep the investigation tractable, we identified three events from each project using the following procedure. For a project:

1) We compute the variance in the number of abandoned implementation (see Section III) files per modules over time, for each module.
2) We select the three modules with the highest variance (excluding the General module).
3) For each module, we select the period with the highest absolute increase in number of abandoned files.

We excluded the last period of Aperreo because it features a major refactoring that threatened the usefulness of our selection. Table III summarizes the resulting events. In the table, the Abandon. column lists the number of module files that became abandoned during the period and the total number of module files abandoned in the period. The Module Size column shows the total number of files in the module. The Lea. column lists the number of leavers that contributed to at least two files of the module.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Period (End)</th>
<th>Abandon. (Period/Total)</th>
<th>Module Size</th>
<th>Lea.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aperreo–core</td>
<td>Oct 2014</td>
<td>45/46</td>
<td>214</td>
<td>1</td>
</tr>
<tr>
<td>Aperreo–klap</td>
<td>Oct 2014</td>
<td>2/8</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>Aperreo–x509</td>
<td>Oct 2014</td>
<td>8/99</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>Assimp–pyassimp</td>
<td>Nov 2008</td>
<td>6/6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Assimp–jassimp</td>
<td>Aug 2012</td>
<td>37/37</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td>Assimp–main</td>
<td>Aug 2014</td>
<td>8/16</td>
<td>319</td>
<td>1</td>
</tr>
<tr>
<td>Chromium–tools</td>
<td>Jul 2014</td>
<td>84/170</td>
<td>1312</td>
<td>7</td>
</tr>
<tr>
<td>Chromium–components</td>
<td>Jan 2015</td>
<td>90/135</td>
<td>1333</td>
<td>2</td>
</tr>
<tr>
<td>Chromium–browser</td>
<td>Jul 2015</td>
<td>109/404</td>
<td>4965</td>
<td>11</td>
</tr>
<tr>
<td>Gimp–app</td>
<td>May 2005</td>
<td>27/30</td>
<td>305</td>
<td>1</td>
</tr>
<tr>
<td>Gimp–core</td>
<td>Feb 2013</td>
<td>15/20</td>
<td>350</td>
<td>1</td>
</tr>
<tr>
<td>Gimp–widgets</td>
<td>Feb 2013</td>
<td>33/33</td>
<td>391</td>
<td>1</td>
</tr>
<tr>
<td>Gitlab–db</td>
<td>Aug 2013</td>
<td>26/30</td>
<td>157</td>
<td>1</td>
</tr>
<tr>
<td>Gitlab–features</td>
<td>Aug 2013</td>
<td>6/10</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>Gitlab–spec</td>
<td>Nov 2013</td>
<td>47/47</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>Kodi–filesystem</td>
<td>Mar 2012</td>
<td>151/158</td>
<td>289</td>
<td>1</td>
</tr>
<tr>
<td>Kodi–viz</td>
<td>Mar 2012</td>
<td>217/451</td>
<td>460</td>
<td>1</td>
</tr>
<tr>
<td>Kodi–x86</td>
<td>Mar 2012</td>
<td>327/350</td>
<td>784</td>
<td>1</td>
</tr>
<tr>
<td>Linux–arch</td>
<td>Apr 2013</td>
<td>83/204</td>
<td>1836</td>
<td>1</td>
</tr>
<tr>
<td>Linux–drivers</td>
<td>Oct 2015</td>
<td>567/1854</td>
<td>7310</td>
<td>122</td>
</tr>
<tr>
<td>Linux–drivers-net</td>
<td>Oct 2015</td>
<td>161/591</td>
<td>2502</td>
<td>46</td>
</tr>
<tr>
<td>Trinity–dep</td>
<td>Feb 2013</td>
<td>27/27</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td>Trinity–sql</td>
<td>Aug 2014</td>
<td>173/610</td>
<td>5623</td>
<td>4</td>
</tr>
<tr>
<td>Trinity–server-game</td>
<td>Feb 2015</td>
<td>13/18</td>
<td>481</td>
<td>4</td>
</tr>
</tbody>
</table>

Contributor Involvement

Like most software repository-based metrics, file abandonment is largely related to the activity of the contributors to a system. Because large projects can have different kinds of contributors, from one-time committers to long-term developers [9], we investigated the impact of such diversity on the interpretation of file abandonment events.

For each event, we extracted the set of leavers who played a role in the event by selecting all who left the project while owning some lines in at least two different files (to exclude trivial contributions) that became abandoned during the event. For 17 of the 24 events, a single leaver contributed to close to 100% of the files of the event. The Chromium-components event was similar, with one contributor who owned lines in 85 of the 90 abandoned files, and the next largest contributor owning lines in only four files. We chose to treat this event like the previous 17 events. To understand the importance of the contributions of these impactful leavers, Figure 5 shows the commits that were made by each leaver prior to their departure.

The figure immediately reveals two issues with leavers. First, we observe the richness of the phenomenon of departure, previously abstracted as a binary classification by the rule that a contributor becomes a “leaver” after their last commit. For example, contributor neo essentially left the Gimp project over two years prior to the threshold, having made only a handful of commits in the intervening time. The same reasoning can apply, to a lesser extent, to HelvetiX (Gimp) and AllTheKiller (Kodi). At the other end of the spectrum, we
note the sudden departure of serya, ialpert, etc. Second, we observe the significant quantitative differences in involvement. Considering that these contributors were entirely responsible for the detection of a major file abandonment event, the impact of the departure of neo (who committed heavily for five years) and that of scholza, who made a single contribution and left, is incomparable. We further investigated this intriguing pattern of immediate abandonment, and report our findings under Ownership Acquisition Events, below. In our case, six impactful leavers participated in their projects for less than a year, three of them for even less than a quarter.

**Contributor Involvement Threat:** Interpretation of metrics that are derived from the activity of a few contributors may be significantly impacted by the actual involvement of these contributors.

We also analyzed the contributor activity for the six remaining events but found that they provided no additional insight.

One additional threat to validity when analyzing contributions to software repositories is email aliasing, i.e., the use of different emails by the same developer. This, however, is a well-known threat to validity [10] that is routinely controlled for through the use of a dealiasing procedure (the heuristic-based semi-automated clustering of email addresses that are presumed to belong to the same person). Such a dealiasing procedure was used in the original studies [4], [6] and reused for the present work. Because this threat is already well documented, and because it directly compounds the other threats presented in this paper, we do not include it as part of our contribution.

**Abandonment Evolution**

Abandonment is a normal part of the evolution of a software system, as developer turnover is almost inevitable. However, a metric to quantify knowledge loss within a period suggests a sudden, abnormal, and deleterious event. The so-called bus factor metric is an example, which evokes imagery of suddenness (in the form of a traffic accident). Yet in practice, the abandonment of a module may grow slowly, eventually passing the detection threshold. This situation poses a risk to the detection of the events, if several files simply happen to exceed the detection threshold during the same period.

To investigate this possibility, we plotted the evolution of the proportion of abandoned lines in the files involved in the 24 events under investigation. We show two representative patterns. For some of the events, such as Linux-drivers-net (Figure 6a), the ratio of abandoned lines in the module slowly grows, with occasional steps. In this case, the event may artificially be the point where the abandonment happens to cross the arbitrary 90% threshold. This contrasts with other events, such as Apero-ldap (Figure 6b), where all files in the module grow from an abandonment rate of less than 10% to over 90% from one period to the next.

The incremental accretion of abandoned files (as opposed to lines) also poses a threat to the detection (or prediction) of loss events. When files are gradually abandoned in a module and never recovered, attempts to detect events based on discontinuities or variance in the data will not succeed despite the fact that the concept of interest (knowledge loss) will be present.

**Quantization Threat:** Using thresholds to classify development artifacts according to property values may create events where none exist. Conversely, progressive software evolution may make concepts of interest undetectable.

**Ownership Acquisition Events**

When analyzing the contribution history of leavers and the abandonment evolution, we observed abnormal patterns of ownership acquisition. Upon further investigation, we noted that some abandonment events were artifacts of unusual ownership acquisition by contributors. The most salient example is the jassimp module of Assimp, which was added to the
system by a contributor who immediately left. Therefore, the whole module appears to be abandoned all at once, yet this does not constitute a loss of the knowledge previously present in the system, especially in the case of bulk imports, migration from another repository, or other similar types of contributions.

A more subtle example relates to Gitlab-spec. This event involves files that were present in the module for a long time, with very low abandonment rates, which suddenly jumped to over 90% in one period. At first glance, this event appears to be similar to the Aperseo-Idap event. However, looking at the evolution of the ratio of lines that were attributed to the leavers in the files involved in the event (Figure 6c), we can clearly see a jump from near 0% to high values in the period just before the abandonment event. This observation leads to the idea that the event is simply the consequence of a punctual, unusual, and large ownership acquisition event. This can be due, for example, to a minor contributor doing a simple automated refactoring that results in numerous line changes, which would thus create large amounts of artificial ownership for the contributor. In such cases, the event is a false positive.

**Exceptional Action Threat:** Subsets of the data that are presumed to be relevant to a construct of interest may only be an artifact of tool-supported activities, such as reverting deleted files or reformattting code files.

**Cohesion of Abandoned Files**

We concluded our micro-analysis of file abandonment events by investigating the relation between the files that were detected to be abandoned as part of an event.

Some of the files that are associated with an event immediately emerged as representing the loss of a cohesive unit of functionality of the system, even to an outside observer. For example, the Assimp-pyassimp and Assimp-jassimp events represent the complete abandonment of the entire ports of the library to Python and Java, respectively.

In contrast, other events consist of a large number of files with no obvious relation other than being located in the same sub-directory. In our case, if the proportion of abandoned files within a folder is relatively low, the knowledge at risk is mitigated by the other contributors who have contributed to related files from the same folder.

Furthermore, even among events that represent the abandonment of a cohesive unit of functionality, the consequences may differ depending on the architecture of the system. The abandonment of functionalities on which many other parts of the system depend will likely have a larger impact than the abandonment of functionalities that do not affect any others, such as independent plug-ins or library wrappers for other programming languages.

**Architectural Sensitivity Threat:** The contribution of different artifacts in a software repository to a construct may be a function of the architecture of the system, which is rarely formally documented in a manner that lends itself to automatic analysis.

V. Threat Assessment and Mitigation Strategies

Our case study of the detailed assessment of the validity of the Knowledge at Risk metric exposes how detailed aspects of software development history can be overlooked when aggregating data from software repositories. To help derive clear implications from this study, we organized our findings in terms of eight major threats to the validity of software process constructs that are to be captured from data aggregation.

Table IV presents these threats together with an assessment question and an example of aggregation that is exposed to the threat. The table is thus contributed as an assessment instrument for metrics. The instrument is designed so that if the answer to any of the questions is affirmative, the metric is subject to the threat and the corresponding data or construct should be carefully reviewed to assess the extent of the risk to validity. We emphasize that a threat is not a flaw: a positive answer may not point to an invalid inference if proper measures are in place to control for the risk of bias. However, mitigation factors should not be accidental, and the role of the assessment instrument is to support the systematic review of data aggregation for software repositories.

In terms of mitigation, we offer the following insights, derived from our investigation and subsequent reflection.

The Fragility Threat is not limited to software repositories, as computing probabilities for rare events (e.g., 100-year floods) relies on strong assumptions about the conditions in which the phenomenon is observed and involves unknown amounts of uncertainty when these assumptions are violated. As part of an idea he called “black swan”, Taleb has argued extensively that empirically-derived probabilities for extreme events are not reliable\(^\text{2}\). The particular risk with software repositories is that the richness of the available data makes it possible to easily compute past frequencies for any number of rare event types, from file abandonments to very large commits, massive test failures, etc. In accordance with Taleb’s philosophy, our recommendation is simply to avoid using the past as a model for the future when rare events are involved.

For the File Content threat, the self-evident but simplistic implication is to take the content of the file into account. Although weighting file-based metrics according to their length is a straightforward first step [6], other factors come into play that are project-specific and thus less easily controlled (such as the presence of constant declarations, cloned build scripts, etc.).

The threat of file content dependence is also closely related to the File Role and Comment threats. The File Role threat is especially problematic because the role of a file may be impossible to confidently detect without insider knowledge (e.g., whether a large file is a cloned-and-owned library or a large initial commit). To mitigate the File Role threat, approximate techniques such as analysis of outliers (e.g., large commits [12]) and explicit detection heuristics can serve as a

\(^2\)This statement applies to natural events with probabilities computed from empirical data. For artificial systems such as games and lotteries, it is obviously possible to reliably compute probabilities for any event.
starting point. The Comment threat directly compounds the File Content threat by further degrading the relation between the concept of a line in a file and an amount of software code. The Comment threat can be mitigated by parsing files for comments and accounting for the resulting ratio, as we did in this study. However, this mitigation is only partial, since comments can also represent different constructs, such as conversations between developers [13], [14], or dead code. For applicable metrics, for a strong mitigation of the Comment threat, one would also need to implement heuristics to classify comments, similarly to the content of messages in developer mailing lists [15].

Conceptually, the Contributor Involvement threat is similar to that of file content, in that contributors, like files, should not be considered identical atomic data points. The threat can thus be controlled through characterization of contributors.

For the Quantization Threat, for certain analyses, it may be possible to detect and leverage “natural” periods in the data [16], or at least implement discontinuity detection algorithms to avoid artificially severing continuously evolving variables. When a threshold is inevitable due to the nature of the construct (as is the case for file abandonment), enhancing threshold detection with a hysteresis mechanism will at least eliminate cases where a variable close to a threshold creates artificial discontinuities by repeatedly crossing the threshold back and forth.

By their nature, the Architectural Sensitivity and Exceptional Action are difficult to avoid algorithmically, and the best defense may simply be diligent investigation of the underlying data and additional extended case studies of specific data items [17].

### VI. Threats and Limitations

Even a study on threats to validity cannot escape the grips of its own threats and limitations.

First, as a necessary way to analyze and visualize the huge amount of data we investigated, we employed various classification rules that are based on assumptions (e.g., to classify files as tests). Although we experimentally designed the thresholds and other rules to be as effective as possible, the classifications have their own error rates. The two main mitigating factors for these threats are that (a) we released our entire data set, so that the accuracy of the classification can independently be assessed, and (b) none of the threats (outcomes) is directly based on a single classification result.

Second, as discussed in Section II, focusing on some subset of the data requires a systematic procedure to select the data to put under the magnifying lens. In the absence of reliable architectural documents, we designed a procedure that is based on the existing directory structure. Other procedures would have resulted in different data being selected for the micro-analysis. The potential of a false positive is mitigated by our experimental approach, where we used the data to illustrate potential threats, as opposed to making claims about their presumed frequency. The main risk with our procedure is the converse: that we may have missed a threat worth reporting because the data that exemplified it was distributed across module boundaries.

Finally, we generalize to other software repository-based metrics. Given our case-study inspired research methods, the logic of inference is analytic rather than statistical [7]. We propose threats that apply to a specific context, with a specific description of the context which we aimed to capture as the assessment question in Table IV. The threat with analytic generalization is that an observed phenomenon may be unique to the case studied. In our case we mitigate this risk by orienting the outcomes in terms of general repository mining concepts (files, lines of code), as opposed to concepts specific to the case (i.e., knowledge at risk).

### VII. Related Work

We review work related to knowledge loss assessment as an illustration of the role and importance of process metrics derived from repository data, and then discuss work on research methods that offer perspectives that are complementary to our catalog of threats.
Identifying which stakeholders have knowledge about modules within a codebase has long been of interest in the software engineering community. To identify stakeholders with relevant experience, Mockus and Herbsleb [18] developed the idea of Experience Atoms (EAs), i.e., completed development tasks (e.g., commits). Since EAs can also be associated with modules, the EA concept can identify stakeholders with expertise in a module. Challenging the hypothesis underlying EAs, Fritz et al. [19] found that developer commit activity was not strongly associated with module knowledge. Indeed, Negara et al. [20] argue that solely using version control records may underestimate the amount of knowledge of the code a developer has, since many of their interactions with a system are not recorded. Hence, more recent work replaces or augments change-based EAs with activities derived from API usage [21], IDE builds [22], IDE interactions [23], [24], and participation in peer code review [25].

Consolidation of knowledge within individuals has been observed to share a relationship with software quality. Rahman and Devanbu [26] found that developers with deep local experience produced code that was less frequently implicated in future defect fixes than that of developers with a breadth of general experience. Similarly, Bird et al. [27] found that the degree to which a module is predominantly developed by a single author shared a strong decreasing relationship with the likelihood of the module containing post-release defects.

By the same token, however, knowledge consolidation exposes an organization to the risk of knowledge loss because of developer turnover. For example, the degree to which an organization is exposed to turnover-induced knowledge loss is colloquially operationalized using the “bus” factor, which is the minimum number (or ratio) of developers who would need to leave the organization to stall development progress. The lower the bus factor value for an organization, the greater the organizational exposure to turnover-related risk of tacit knowledge being lost. Recent work has attempted to estimate this factor in large open source organizations [28]–[30].

More recent work has focused on identifying areas of a codebase that are at risk. As part of the seminal work on knowledge loss, Izquierdo-Cortazar et al. [8] developed the concept of orphaned code, i.e., code that was contributed by developers who are no longer in the organization, to study knowledge loss and how open source organizations recover from it. Rigby et al. [4] proposed Knowledge-at-Risk (KaR), an adapted version of the Value-at-Risk measure from the financial analysis domain. They used KaR to quantify turnover-induced knowledge loss and proposed strategies to mitigate organizational exposure in Chromium and a project developed by Avaya. Nassif and Robillard [6] then extended the KaR measure to allow for changes in the time period and the weight of files, and used it to study turnover-induced knowledge loss in seven additional open source projects.

META-ANALYSES OF REPOSITORY MINING

Software repositories are noisy and incomplete sources of software data. Herbsleb and Grinter [31] found that (collocated) development teams often make project decisions in face-to-face (offline) meetings. Traces of these decisions are not present in software repositories. Moreover, Aranda and Venolia [17] found that issue trackers and version control repositories only contained partial or even erroneous information with respect to the resolution of issues at Microsoft.

In addition, others have sounded the alarm about the risks of taking at face value results that have been mined from software repositories. In particular, Bird et al. [32] warned of the perils of mining Git repositories. In contrast to the present study, their work focused on the technical aspects of the Git source code management system. Others have pointed out that version control data includes non-essential [33], tangled [34], and non-meaningful changes [35]. The presence of these types of noise in repository data further threaten the meaning of the line ownership data upon which aggregated measures are based. In a different vein, Kalliamvakou et al. [36] warn against the indiscriminate sampling of projects from GitHub. This threat is, however, orthogonal to the threats that we identified, which apply to data aggregation for a given project.

Threat assessment and mitigation mechanisms have also been recently proposed. For example, Nagappan et al. [37] propose a sample coverage measure to quantify the threat of external validity to a population of interest. Munaiah et al. [38] propose Reaper, a tool to assess the likelihood that a GitHub repository contains an “engineered” software project. Similar to these past efforts, we propose an evaluation instrument to assess threats to construct validity that may be present when aggregating software repository data.

VIII. CONCLUSION

Countless benefits can be derived from the analysis of software repositories, as actionable insights can be produced at little cost once an analysis technique is in place. However, care must be taken to ensure that insights are based on a valid interpretation of the data.

To appreciate first hand the risk to the relation between a metric and the construct it is intended to represent, we conducted an in-depth critical review of the file abandonment computations described in previous work. The case study raised our awareness of the validity threats for metrics derived from aggregated software repository data. We organized our findings in terms of eight major threats to validity to consider when designing new software repository mining studies or techniques, and report these threats as an instrument with assessment questions and examples. The instrument can be used both to evaluate existing metrics and to help design further repository-based metrics and mining techniques.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for constructive feedback. This work was funded by NSERC.