A Novel Approach for Estimating Truck Factors

Guilherme Avelino*†, Leonardo Passos‡, Andre Hora* and Marco Tulio Valente*
*ASERG Group, Department of Computer Science (DCC)
Federal University of Minas Gerais (UFMG), Brazil
Email: {gaa, mtov, hora}@dcc.ufmg.br
† Department of Computing (DC)
Federal University of Piaui (UFPI), Brazil
‡University of Waterloo, Canada
Email: lpassos@gsd.uwaterloo.ca

***Abstract*—Truck Factor (TF) is a metric proposed by the agile community as a tool to identify concentration of knowledge in software development environments. It states the minimal number of developers that have to be hit by a truck (or quit) before a project is incapacitated. In other words, TF helps to measure how prepared is a project to deal with developer turnover. Despite its clear relevance, few studies explore this metric. Altogether there is no consensus about how to calculate it, and no supporting evidence backing estimates for systems in the wild. To mitigate both issues, we propose a novel (and automated) approach for estimating TF-values, which we execute against a corpus of 133 popular project in GitHub. We later survey developers as a means to assess the reliability of our results. Among others, we find that the majority of our target systems (65%) have TF $\leq$ 2. Surveying developers from 67 target systems provides confidence towards our estimates; in 84% of the valid answers we collect, developers agree or partially agree that the TF's authors are the main authors of their systems; in 53% we receive a positive or partially positive answer regarding our estimated truck factors.**

***Index Terms*—Code Authorship, GitHub, Truck Factor**

I. Introduction

A system's truck factor (TF) is defined as "*the number of people on your team that have to be hit by a truck (or quit) before the project is in serious trouble*" [1]. Systems with a low truck factor spot strong dependencies towards specific personnel, forming knowledge silos among developer teams. If such knowledgeable personnel abandon the project, the system's lifecycle is seriously compromised, leading to delays in launching new releases, and ultimately to the discontinuation of the project as whole. To prevent such issues, comprehending a system's truck factor is a crucial mechanism.

Currently, the existing literature defines truck factor loosely. For the most part, there is no formal definition of the concept, nor means to estimate it. The main exception we are aware of stems from the work of Zazworka et al. [2]. Their definition, however, as well as follow-up works [3], [4], is not backed by empirical evidence from real-world software systems. Stated otherwise, TF-estimates, as calculated by Zazworka's approach, lack reliability evidence from systems in the wild.

Our work aims to improve the current state of affairs by proposing a novel approach for estimating truck factors, backed up by empirical evidence to support the estimates we produce. In particular, we define an automated workflow for TF-estimation for which we apply to a target corpus comprising 133 systems in GitHub. In total, such systems have over 373K files and 41 MLOC; their combined evolution history sums to over 2 million commits. By surveying and analyzing answers from 67 target systems, we evidence that in 84% of valid answers developers agree or partially agree that the TF's authors are the main authors of their systems; in 53% we receive a positive or partially positive answer regarding our estimated truck factors.

From our work, we claim the following contributions:

1) A novel approach for estimating a system's truck factor, as well as a publicly available supporting tool.[1]
2) An estimate of the truck factors of 133 GitHub systems. All our data is publicly available for external validation,[2] comprising the largest dataset of its kind.
3) Empirical evidence of the reliability of our truck factor estimates, as a product of surveying the main contributors of our target systems. From the survey, we report the practices that developers argue as most useful to overcome a truck factor event.

We organize the remainder of the paper as follows. In Section II we present a concrete example of truck factor concerns in the early days of Python development. In Section III we present our novel approach for truck factor estimation, detailing all its constituent steps. Next, Section IV discusses our validation methodology, followed by the truck factors of our target systems in Section V. We proceed to present our validation results from a survey with developers (Section VI), further discussing results in Section VII. We argue about possible threats in Section VIII. We present the related work in Section IX, concluding the paper in Section X.

II. Truck Factor: An Example from the Early Days of Python

"What if you saw this posted tomorrow: Guido's unexpected death has come as a shock to us all. Disgruntled members of the Tcl mob are suspected, but no smoking gun has been found..."—Python's mailing list discussion, 1994.[3]

[1] https://github.com/aserg-ufmg/Truck-Factor
[2] http://aserg.labsoft.dcc.ufmg.br/truckfactor
[3] http://legacy.python.org/search/hypermail/python-1994q2/1040.html

Years before the first discussions about truck factor in eXtreme programming realms,[4] this post illustrates the serious threats of knowledge concentration in software development. By posting the fictitious news in Python's mailing list, the author, an employee at the National Institute of Standards and Technology/USA, wanted to foster the discussion of Python's fragility resulting from its strong dependence to its creator, Guido van Rossum:

"I just returned from a meeting in which the major objection to using Python was its dependence on Guido. They wanted to know if Python would survive if Guido disappeared. This is an important issue for businesses that may be considering the use of Python in a product."

Fortunately, Guido is alive. Moreover, Python no longer has a truck factor of one. It has grown to be a large community of developers and the fifth most popular programming language in use.[5] However, the message illustrates that Python was, at least by some, considered a risky project. As knowledge was not collective among its team members, but rather concentrated in a single "hero", in the absence of the latter, discontinuation was a real threat, or at minimum, something that could cause extreme delays. Projects with low truck factor, as Python was in 1994, face high risk adoption, discouraging their use.

To facilitate decision making, it is crucial to metrify the risks of project failure due to personnel lost. This information serves not only business managers assessing early technology adoption risks, but also maintainers and project managers aiming to identify early knowledge silos among development teams. Timely action plans can then be devised as a means to prevent long-term failure. In that direction, reliable estimations of a project's truck factor is a must.

III. Proposed Approach

We calculate the truck factor of a target system by processing its evolution history. We assume the latter to be managed by a version control system, in addition to having access to a local copy of the repository of the target subject.

Our approach comprises five major steps—see Figure 1. Step 1 checkouts the latest point in the commit history, listing all the source files therein. Step 2 handles possible aliases among developers, *i.e.*, cases where a single developer has multiple Git users. Step 3 traces the history of each source code file. From such traces, step 4 defines the authors in the system, as well as their authored files. Authorship, in this case, is not a strict notion. Stated otherwise, authorship is not a matter of who creates a file. Rather, authorship is a statement of who will be able to maintain a file from the latest system snapshot onward. This may comprise the creator of the file (original author), as well as other developers (co-authors) who significantly contributed with changes to a file after its creation. With the list of authors and their authored files, step 5 estimates the truck factor of the entire system.

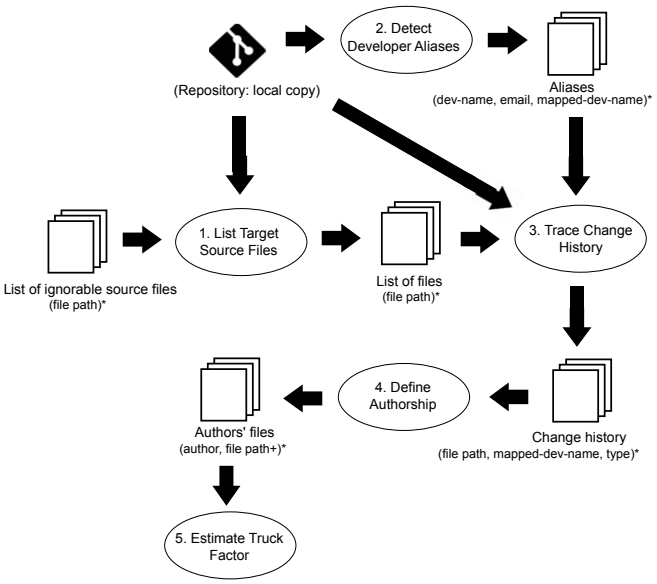


Fig. 1. Proposed approach for truck factor calculation

We realize the given process to automatically estimate the truck factor of projects whose evolution is managed by Git. In the following, we detail the realization of each step.

A. Realization

Step 1: List Target Source Files. To obtain the list of target files, we first switch to the master branch of the target repository, checking out its latest commit. We then enumerate the path of all source files of the given snapshot, excluding all other file types (*e.g.*, files representing documentation, images, examples, etc), as well as the files listed in the ignorable source file list, given as input. We also discard source files associated with third-party libraries (*i.e.*, files that are not developed in the system under analysis). Our decision is conservative. An existing survey from JavaOne'14[6] reports that nearly two-thirds of polled senior IT professionals have Java applications with half of their code coming from third-party sources. Thus, if developers store third-party code in the system's main Git repository (*e.g.*, as backup, to facilitate build, etc), and third-party code is as large as the poll suggests, truck factor estimates are likely to be significantly affected.

To exclude third-party code, one must be able to identify it in the first place. As such, we employ Linguist,[7] an opensource tool from GitHub. Linguist is actively developed, and it is constantly being updated by the GitHub community to include new pattern matching rules to identify third-party file names. Linguist's original goal is to detect the programming language of GitHub projects—as in our case, this is sensitive to external code.

Step 2: Detect Developer Aliases. Each user in Git is a pair (*dev-name*, *email*)—*e.g.*, ("Bob Rob",

[4]http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/
[5]http://www.tiobe.com/index.php/tiobe_index
[6]http://tinyurl.com/javaone14-survey
[7]https://github.com/github/linguist

"bob.rob@example.com"). It happens, however, that a single developer may be associated with many Git user accounts, leading to *developer aliases*. As such, this step first extracts all dev-names and their associated emails.[8] From the listing, we group users with the same email, but possibly with different associated dev-names. Additionally, we unify similar dev-names, but with different emails. In the latter case, we employ the Levenshtein distance [5], with a threshold of at most one. This means that we allow at most one single-character insertion, deletion or substitution to claim two different dev-names as referring to the same developer. For example, if "Bob.Rob" and "Bob Rob" are two dev-names, we consider them as the same developer, as a single substitution is needed to make their names identical. As a result of alias detection, step 2 outputs a mapping from Git users to a single developer name (*mapped-dev-name*).

Step 3: Trace Change History. This step traces the evolution history of each target file, taking as input the results of the previous two steps. To perform the tracing, we collect the system's commits using the `git log --find-renames` command. This command returns all commits of a repository and identifies possible file renames. We process each commit, extracting three pieces of information: (i) the path of the file we are collecting the trace; (ii) the *mapped-dev-name* of the developer performing the change; and (iii) the type of the change—file addition, file modification, or file rename.

Step 4: Define Authorship. Given the change traces of each file in the target snapshot of the project at hand, this step defines the author list of each file. Different alternatives could be used as a means for determining authorship—*e.g.*, [6]–[11]. Among those, we chose the *degree-of-authorship* (DOA) metric [10], [11], which we normalize after calculation. Given a file f with path f_p, the degree-of-authorship of a developer d whose Git user has been mapped to m_d is given by:

$$DOA(m_d, f_p) = 3.293 + 1.098 \times FA(m_d, f_p) + 0.164 \times DL(m_d, f_p) - 0.321 \times \ln(1 + AC(m_d, f_p))$$

From the equation, DOA depends on three factors: (i) first authorship (FA): if m_d originally created f, *FA* is 1; otherwise it is 0; (ii) number of deliveries (DL): number of changes in f made by m_d; and (iii) number of acceptances (AC): number of changes in f made by any developer, except m_d.

The model assumes FA as the strongest predictor of authorship. Recency information (DL) positively contributes to authorship, but with less importance. In contrast, other developers' changes (AC) decrease one's DOA, although at a slower pace. The weights we use in the DOA model stem from empirical experiments performed elsewhere [11].

DOA has three major advantages in comparison to other approaches: (i) assuming the weights of the model to be general, DOA does not require a training set; (ii) DOA does not require monitoring editing activities as developers maintain different files (*e.g.*, as in [9]); (iii) instead of considering all developers changing a file as its authors (*e.g.*, as in [2]), DOA weights contributions differently, accounting for both changes of a developer in a file (increases DOA), as well as the changes performed by others (decreases DOA).

[8]For instance, by issuing `git log | grep "Author:" | sed 's/^.*:\s\+//;s/\s\+</;/;s/>$//'| sort | uniq`

Algorithm 1: TRUCK FACTOR ALGORITHM.

```
   Input: List of authors' files A
   Output: System truck factor
1  begin
2      F ← getSystemFiles(A);
3      tf ← 0;
4      while A ≠ ∅ do
5          coverage ← getCoverage(F, A);
6          if coverage < 0.5 then
7              break;
8          end
9          A ← removeTopAuthor(A);
10         tf ← tf + 1;
11     end
12     return tf;
13 end
```

Once we know the DOA-values of all files changed by previously mapped developers, we proceed to normalize results. A normalized DOA-value ranges from 0 to 1. We grant 1 to the developer with the highest absolute DOA among all developers that worked on f; altogether, we consider a developer as an author of a file if its resulting normalized DOA is greater than a threshold k and its absolute DOA is not lower than a value m. Currently, k and m stands as configurable parameters in our approach. As we show in Section IV-B, $k = 0.75$ and $m = 3.293$ seem to provide good results. As a result of this step, we output a list of associations from authors (*mapped-dev-names*) to their related authored files.

Step 5: Estimate Truck Factor. Taking a list A of authors (*mapped-devs*) and their associated authored files (one or more file paths), this step estimates the system's truck factor. Our estimation relies on a coverage assumption: a system will face serious delays or will be likely discontinued if its current set of authors covers less than 50% of the current set of files in the system. Following such assumption, our truck factor estimation algorithm implements a greedy heuristic—see Algorithm 1. Starting with a truck factor of zero, we iterate over the authors' file list A (lines 4–11), verifying at each iteration whether the current authors' coverage is below 0.5 (line 6). If so, we stop the iteration—maintenance is likely to be hampered; otherwise, we remove the top author from A (line 9), increasing truck factor by one (line 10). The top author in a given iteration is the *mapped-dev* authoring the highest number of files in A.[9] Whenever A shrinks, another iteration follows, provided A is not empty. This process continues until A becomes empty or coverage is less than 0.5.

IV. VALIDATION METHODOLOGY

To validate our approach, we select 133 systems from GitHub. For each target system, we estimate its truck factor.

[9]This is obtained by finding the entry $e_i = (a_i, \textit{filepath-list}_i) \in A$ s.t. $\nexists\ e_j = (a_j, \textit{filepath-list}_j) \in A \wedge e_j \neq e_i \wedge |\textit{filepath-list}_j| > |\textit{filepath-list}_i|$. If there exist more than one top author, we just take the first one we find.

This section details our corpus selection and how we setup our approach for estimating truck factors for our chosen subjects. We also discuss how we survey developers as a means to validate our estimates and get further insights.

A. Selection of Target Subjects

To select a target set of subjects, we follow a procedure similar to other studies investigating GitHub [12]–[15]. First, we query the programming languages with the largest number of repositories in GitHub. We find six main languages (L): JavaScript, Python, Ruby, C/C++, Java, and PHP. We then select the 100-top most popular repositories within each target language. Popularity, in this case, is given by the number of times a repository has been starred by GitHub users. Considering only the most popular projects in a given language (S_ℓ), we remove the systems in the first quartile (Q_1) of the distribution of three metrics, namely number of developers (n_d), number of commits (n_c), and number of files (n_f). After filtering out subjects in Q_1, we compute the intersection of the remaining sets. From the previous steps, we get an initial set of prospective subjects T^0. Formally,

$$T^0 = \bigcup_{\ell \in L} T^0_{n_d}(\ell) \cap T^0_{n_c}(\ell) \cap T^0_{n_f}(\ell)$$

where

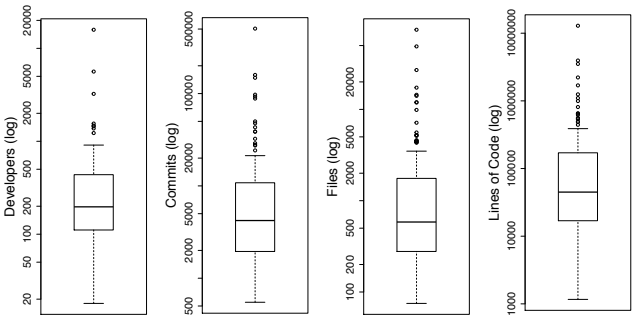
$$T^0_{n_d}(\ell) = S_\ell - Q_1(n_d(S_\ell))),\ T^0_{n_c}(\ell) = S_\ell - Q_1(n_c(S_\ell))),$$
$$T^0_{n_f}(\ell) = S_\ell - Q_1(n_f(S_\ell)))$$

From T^0, we determine a new subset T^1 including only the systems whose repositories stem from a correct migration to GitHub. Specifically, we remove systems with more than 50% of their files added in less than 20 commits—less than 10% of the minimal number of commits we initially considered. This evidences that a large portion of a system was developed using another version control platform and the migration to GitHub could not preserve the original version history. From the resulting set of prospective subjects ($|T^1| = 135$), we manually inspect the documentation in each repository to identify and eliminate duplicate subjects. Our inspection shows raspberrypi/linux and django/django-old as duplicate cases. The first, despite not being a fork, is very similar to torvalds/linux; in fact, it is a clone of the Linux kernel, with extensions supporting RaspberryPi-based boards. The second is an old version of a repository already in T^1.

After excluding raspberrypi/linux and django/django-old, we are left with 133 subjects (T^2), which represent the most important systems per language in GitHub, implemented by teams with a considerable number of active developers and with a considerable number of files. Table I summarizes the characteristics of the repositories of our chosen subjects. Ruby is the language with more systems, 33 in total. The programming language with less systems is PHP, with 17 projects. Accounting all our chosen subjects, their latest snapshots accumulate over 373K files and 41 MLOC; their combined evolution history sums to over 2 million commits. Our targets also have a large community of contributors, accumulating to over 60K developers. Figures 2(c)–2(d) depict each distribution.

TABLE I
TARGET REPOSITORIES

Language	Repos	Devs	Commits	Files	LOC
JavaScript	22	5,740	108,080	24,688	3,661,722
Python	22	8,627	276,174	35,315	2,237,930
Ruby	33	19,960	307,603	33,556	2,612,503
C/C++	18	21,039	847,867	107,464	19,915,316
Java	21	4,499	418,003	140,871	10,672,918
PHP	17	3,329	125,626	31,221	2,215,972
Total	**133**	**63,194**	**2,083,353**	**373,115**	**41,316,361**



(a) Developers (b) Commits (c) Files (d) LOC

Fig. 2. Target subjects

B. Setting up Inputs

Our approach requires as input a listing of ignorable source files of a system, in addition to a tweak of the DOA thresholds (k and m). Next, we detail how we set both inputs.

List of Ignorable Source Files. To create the list of ignorable files, we manually inspect the first two top-level directories in each target repository, seeking to find third-party libraries undetected by Linguist. Also, as Linguist is architecture and system agnostic, we look for plugin-related code in systems with a plugin-based architecture. As with third-party code, plugins may highly influence a system's truck factor. For instance, in the Linux kernel, driver plugins are the most common feature type [16]; since driver features generally denote optional features targeting end-user selection, the kernel itself is independent from them. In the case of torvalds/linux, we exclude all driver-related code, which is, for the most part, inside the `driver` folder of the Linux kernel source code tree.[10]

In addition to the Linux kernel repository, two other systems have a large amount of plugin-related code: Homebrew/homebrew and caskroom/homebrew-cask. Homebrew is a package manager in Mac OS for handling the installation of different software systems. Its implementation allows contributors to push new formulas (automated installation recipes) to the system's remote repository, leading to thousands of formulas. As an extensible software system, Homebrew has one of the largest base of developers on GitHub (more than 5K developers, as of July 14th, 2015). Considering all its formu-

[10]Specifically, we identify all driver-related code by executing a specialized script from G. Kroah-Hartman, one of the main developers of the Linux kernel. Available at https://github.com/gregkh/kernel-history.

las, Homebrew's TF, as computed by our heuristic, is 250. After excluding the files in folder `Library/Formula`, however, HomeBrew's truck factor reduces to 2. This clearly evidences the sensitivity of TF-values in the face of external code. As for caskroom/homebrew-cask, we ignore its `Casks` directory.

In total, our list of ignorable files excludes 10,450 entries.

Setting DOA Thresholds. To find suitable thresholds, we manually inspect a random sample of 120 files stemming from the six top-most popular systems in our target corpus (T^2), one for each target language we account for. This results in files from mbostock/d3 (JavaScript), django/django (Python), rails/rails (Ruby), torvalds/linux (C/C++), elasticsearch/elasticsearch (Java), and composer/composer (PHP). We then compute the normalized DOA-values for each developer contributing at least one commit changing a file in our sample. Initially, we note that normalized DOA-values below 0.50 lead to doubtful authorships. We measure doubtfulness by contrasting our authorship results with ranks we extract from `git-blame` reports. The latter contains the last developer who modified each line in a file [17]; by ranking developers according to the number of their modified lines, authors are likely to be those with higher ranks. Fixing 3.293 (which corresponds to the constant term in DOA's linear equation) as minimal absolute DOA and resetting the threshold for normalized DOA to 0.75 better aligns results. Specifically, 64% of the authors selected using those thresholds are classified as the top-1 ranked developer from `git-blame`; in 91% of the cases, they are among the top-3 in the ranking list of `git-blame`, whereas 7% lie between the 4th and 8th positions. In only three cases (2%), the authors do not pair with any developer from `git-blame` rankings.

C. Survey Design and Application

After collecting the truck factors of our chosen targets, one of the authors of this paper set to elaborate survey questions seeking to evidence the reliability of our results, as well as an instrument to get further insights. Following best practices in survey design [18], we assure clarity, consistency, and suitability of our questions by running a feedback loop between the survey author and two other authors of this paper, until reaching a consensus among all three. We also perform a pilot study to identify early problems, such as whether our language correctly captures the intent of our questions. From the pilot study, we note few, but important communication issues, which we fix accordingly.

Survey Questions. After our pilot study, we phrase our questions as follows.

> Question 1. Do developers agree that top-ranked authors are the main developers of their projects?

This question seeks to assess the accuracy of our top authorship results. The top-ranked authors of a system are those we remove during the iteration step of our greedy-heuristic (recall Algorithm 1), *i.e.*, those responding for a system's truck factor. Note that we use the term *main developers*, not authors. Our pilot study shows that developers tend to consider the creator of a file as its main author.

> Question 2. Do developers agree that their project will be in trouble if they loose the developers responding for its truck factor?

This question aims to validate our TF estimates. If we receive a positive feedback in this question, we can conclude that code authorship is an effective proxy.

> Question 3. What are the development practices and characteristics that can attenuate the loss of the developers responsible for a system's truck factor?

Our intention here is to reveal the instruments developers see as most effective to circumvent the loss of important developers—*e.g.*, by devising better documentation, codification rules, modular design, etc.

Survey Application. Before contacting developers and applying our survey, we aim at calling their attention by promoting our work in popular programming forums (*e.g.*, Hacker News) and publishing a preprint at PeerJ (https://peerj.com/preprints/1233).

After promotion, we apply the survey by opening GitHub issues in all target projects allowing such a feature (114 out of 133). The choice for issues is twofold: (i) issues foster public discussions among project developers; (ii) issues document all discussions, making them available for later reading. Potential readers include new developers, end-users interested on the target systems, researchers, etc.

Our posting period ranges from July 31th to August 11th, 2015. In the following two weeks, we set to collect answers, respected the deadline of August 25th, 2015. In total, we collect answers from developers of 67 systems. In 37 of those, there is a single answer from a single developer. However, often, the issues include discussions among different project members. For example, in saltstack/salt, we have comments from six developers. In total, we accumulate 170 discussion messages from 106 respondents; 96 messages stem (57%) from the top-10 contributors of the 67 participating projects. Figure 3 characterizes all participants according to their level of project contribution. We get the list of top contributors by consulting the project's statistics as provided by GitHub. To exclude unreliable answers, we discard issues that do not have a single answer from a top-10 project contributor—five issues in total. Thus, we are left with 62 participating systems, as we have one issue per system. Among the issues that we do not exclude from analysis, we find 96 different respondents. Some messages are quite detailed. For instance, a message in an issue in elastic/elasticsearch contains 1,670 words. In fact, according to the respondent, it triggered interesting internal discussions.

Survey Analysis. To compile the survey results, we analyze the discussions of our opened issues. For the first two questions, we classify answers according to four levels: agree, partially agree, disagree, or unclear. The fourth level refers to cases

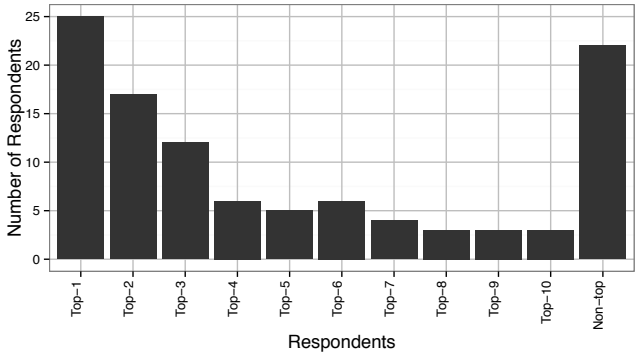


Fig. 3. Respondents profile

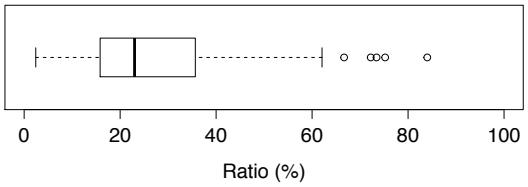


Fig. 4. Proportion of developers ranked as authors

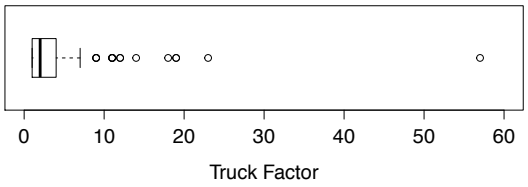


Fig. 5. Systems Truck Factor

where we cannot derive a clear position from an answer. Two authors of this paper independently classified all answers, later crosschecking their results.

As for our third question, we categorize answers to identify common practices and characteristics. Our categorization closely follows grounded theory open-coding principles [19].

V. Truck Factor Estimates

A. Preceding Output

Target List of Source Files (Step 1). Using our input list of ignorable files (see Section IV), as well as the automated exclusion by Linguist, we estimate the truck factor of 243,660 files (33 MLOC)—34% less files than the original set in our subjects. The most frequent kind of files we remove concern JavaScript (5,125), PHP (3,099), and C/C++ (2,049) source files. Decreasing the number of target files decreases the target number of developers (63,193) and commits (1,262,130), a reduction of 28% and 39% w.r.t the original state of our target repositories.

Authorship List (Step 4). By applying the normalized DOA to define the list of authors in each target system, as well as their authoring files, step 4 reveals the proportion of developers ranked as authors—see Figure 4. For most systems, such proportion is relatively small; the first, second, and third quartiles are 16%, 23%, and 36%, respectively. Interestingly, systems with a high proportion of authors usually have support of private organizations. Examples include four of the top-10 systems with the highest author ratio among developers, such as v8/v8 (75%), JetBrains/intellij-community (73%), WordPress/WordPress (67%), and Facebook/osquery (62%). We also detect two language interpreters among the top-10 systems: ruby/ruby (72%) and php/php-src (59%). At the other extreme, there are systems with a very low author ratio—*e.g.*, sstephenson/sprockets (3%) and jashkenas/backbone (2%). Backbone is also an example, with only six authors amongst its 248 developers. These six authors monopolize 67% of commits. A similar situation occurs with sprockets (a Ruby library for compiling and serving web assets): although 61 developers associate to commits in the evolution history, 95% of commits come from two authors only; moreover, 27 developers respond for a single commit modifying a single line of code.

TABLE II
Systems with highest truck factors

System	TF
torvalds/linux	57
fzaninotto/Faker	23
android/platform_frameworks_base	19
moment/moment	19
php/php-src	18
odoo/odoo	14
fog/fog	12
git/git	12
webscalesql/webscalesql-5.6	11
v8/v8	11
Seldaek/monolog	11
saltstack/salt	11
JetBrains/intellij-community	9
rails/rails	9
puppetlabs/puppet	9

B. Results

Figure 5 presents the distribution of the truck factor amongst our subjects. The first, second, and third quartiles are 1, 2, and 4, respectively. Most systems have a small truck factor: 45 systems (34%) have TF = 1 (*e.g.*, mbostock/d3 and less/less.js); in 42 systems (31%), TF = 2 including well-known systems such as clojure/clojure, cucumber/ cucumber, ashkenas/ backbone and elasticsearch/elasticsearch. Systems with high TF-values, however, do exist. Table II presents the top-15 (boxplot outliers)[11] systems with the highest truck factors. Among those, torvalds/linux has TF = 57, followed by fzaninotto/Faker (TF = 23) and android/platform_frameworks_base (TF = 19). Other well-known systems include php/php-src (TF = 18), git/git (TF = 12), v8/v8 (TF = 11), and rails/rails (TF = 9).

[11]In the boxplot, seven outliers have overlapping values, causing them to be plotted above an existing datapoint.

VI. TF Validation: Surveying Developers

We present our survey results from our filtered set of issues and their underlying messages—we only account issues having at least one message from a top-10 project contributor. In total, the answers we analyze stem from 106 respondents, of which 84 are top-10 contributors. The final number of participating systems is 62.

Question 1. Do developers agree that the top-ranked authors are the main developers of their projects?

TABLE III
Answers for Survey Question 1

Agree	Partially	Disagree	Unclear
31 (50%)	18 (29%)	9 (15%)	4 (6%)

Table III summarizes the answers for our first question. Respondents of 31 systems (50%) fully *agree* with our list of main developers. Example agreements:

"Yes, that's me."—developer from bjorn/tiled.

"I think that it is a reasonable statement to make. They have contributed by far the most and paved the way for the rest of us."—developer from composer/composer.

Developers of 18 systems (29%) *partially agree* with our list of top-ranked authors. The main disagreement stems from the historical balance between older and newer developers:

"Yes and no, historically yes, currently no, a team has been picking up the activity, your analysis seems to be biased on capital (existing files) rather than activity (current commits)."—developer from kivy/kivy.

"I would have added @DayS and @WonderCsabo as main developers."—developer from excilys/androidannotations

The latter answer illustrates a situation where we report two top-authors in a target project; the respondent, although agreeing with our suggestion, recommend adding two other developers. The latter two have many recent commits; in contrast, one of the top authors we recommend is no longer active, strengthening the developer's argument. The two top-authors from our degree-of-authorship measures cover 41% and 26% of files, respectively. The two suggested by our respondent account for 9% and 17% (see Figure 6). However, we do note a gradual decrease in the number of authored files by the top developer we suggest, while an increasing trend for one of the two that our respondent recommends.

Developers of nine systems (15%) *disagree* with our list. Six developers indicate that other contributors are now responsible for their projects. Example disagreements include:

"No. TJ has been away from Jade for quite some time now. @ForbesLindesay is considered the main maintainer/developer of Jade."—developer from jadejs/jade.

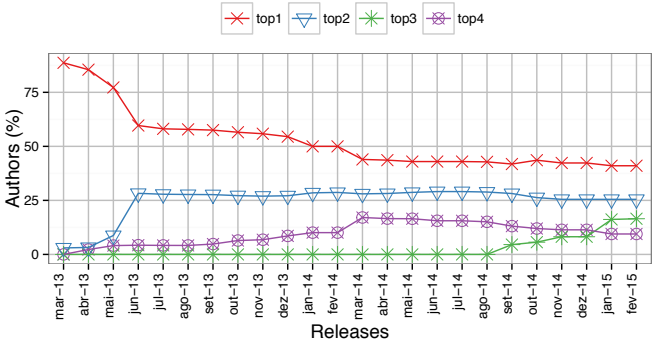


Fig. 6. Percentage of files per author in excilys/androidannotations (top-4 authors)

"No, Burns hasn't been contributing for a while now. I've taken over what he was doing."—developer from backup/backup.

Other disagreements are due to auto-generated code. Finally, a single developer has a negative attitude towards the question, providing us with no insights.

Question 2. Do developers agree that their projects will be in trouble if they loose the truck factor authors?

Table IV summarizes results concerning this question.

TABLE IV
Answers for Survey Question 2

Agree	Partially	Disagree	Unclear
24 (39%)	6 (10%)	27 (43%)	5 (8%)

Developers of 24 systems (39%) *agree* with our truck factor results. Most positive answers are concise, usually a straight "yes" (14 answers). We consider as agreement answers that acknowledge a serious impact to the project if the given developers are to be absent, such as in:

"If both of us left, the project would be kind of unmaintained."—developer from SFTtech/openage.

"Initially, yes. However, given the size of the Grunt community, I believe a new maintainer could be found."—developer from gruntjs/grunt.

"If Wladimir or Pieter left, it would be a serious loss, but not fatal I think."—developer from bitcoin/bitcoin.

Among the positive answers, we find a system that in fact "lost" its single truck factor author—pockethub/PocketHub. The project implements a GitHub Android client, originally released as part of the GitHub platform. A GitHub employee is identified as the system's single author, accounting for 78% of all source files. However, as stated in the repository home page, GitHub no longer maintains the app. The repository is almost inactive, receiving very few commits per month. The last release dates from February 2014 (still as a GitHub project). One developer reports that low community involvement is the

reason for the project's trouble, as there are only three people working on the project and this is not their full time job.

Six answers are *partial agreements*. Examples:

"Somewhat agree. A loss in one area would mean a temporary dip in maintenance of that area until someone else stepped in."—developer from saltstack/salt.

"Not necessarily, there's a long list of both small and significant contributors that had to understand a large piece of the code base to implement a feature or fix."—developer from justinfrench/formtastic.

Developers of 27 systems *disagree* with our TF-values. Six developers (22%) have negative answers to our question, but do not provide further details; 21 developers (78%) justify their answer stating that others could take over the project:

"Backup shouldn't be in trouble... It's an open source project, anyone can start contributing if they want to."—developer from backup/backup.

We find two systems surviving the "lost" of the truck factor authors in our list:

"Coda was the author of the majority of the code. He left the project around a year ago. Some issues were going a long time without resolution, at which point I offered to maintain the project."—developer from dropwizard/metrics.

"Your questions are timely, since Roland [the main author] has already left the project ... and we are not in trouble."—developer from caskroom/homebrew-cask.

In the case of dropwizard/metrics, it has been partially affected by the lost of its single truck factor author, as another developer was able to take over the project. As for caskroom/homebrew-cask, the respondent highlights two factors helping their transition after loosing their single truck factor author: (a) comprehensive documentation; (b) developers ready to transmit the rules and requirements to newcomers.

Question 3. What are the development practices that can attenuate the loss of top-ranked authors?

Table V summarizes answers. Documentation is the practice with the largest number of mentions across answers (36 answers), followed by the existence of an active community (15 answers), automatic tests (10 answers), and code legibility (10 answers). We group practices with a single answer under the *miscellaneous* category—*e.g.*, implementation in an specific programming language, periodic team chats, code reviews, support to classical algorithms, etc. In addition, we received seven "yes/no" answers, which are non-sensical given the nature of the question (not shown).

Although not development practices, having active communities and paid developers appear frequently among the answers we analyze. Both reasons appear as justifications for not concerning with top-authors lost:

"I'd say that the vibrant community is the reason for it."—developer from rails/rails.

"We have a handful of other maintainers and a large body of contributors who are interested in Homebrew's future."—developer from Homebrew/homebrew.

"The people you listed are paid to work on the project, along with a number of others. So if the four of us took off, the project would hire some more people"—developer from ipython/ipython.

TABLE V
PRACTICES TO ATTENUATE THE TRUCK FACTOR

Practice	Answers
Documentation	36
Active community	15
Automatic tests	10
Code legibility	10
Code comments	7
Founding/Paid developers	5
Popularity	5
Architecture and design	4
Shared repository permissions	4
Other implementations	2
Knowledge sharing practices	2
Open source license	2
Miscellaneous	9

VII. DISCUSSION

In this section, we discuss the lessons we learn in our study. We also lay out directions for future research on truck factor measurements and applications.

A. DOA Results

The results produced by the DOA model seem accurate when applied to a large collection of systems. For the first survey question, the developers of 49 systems (84% of the valid answers) agree or partially agree with our results. Despite that, some developers report that the model gives high emphasis on first authorship (FA) events. In the same question of our survey, six developers disagree with our results exactly due to this resilience of the DOA model in transferring authorship from the first author to another one. This applies in systems where a single developer creates the bulk of the code, but later switches role (*e.g.*, project leader or mentor), becoming less active in development activities. In fact, some developers suggest that DOA computation should consider only the most recent development history, *e.g.*, commits performed in the last year. One developer from clojure/clojure explicitly declares that "*if the code is old enough, even the original author will have to approach it with essentially fresh eyes.*"

B. Challenges on Computing Truck Factors

We receive answers for 67 (out of 114) systems. This high response ratio (59%) is certainly a consequence of the importance that developers give to the truck factor concept. By analyzing the answers, we see that developers generally recognize the impact that the truck factor may have in the public

reputation of their systems. However, it is worth noting that estimating this concept automatically has many challenges. A few developers refused to answer our question, stating for example that "*it is an existential, speculative question that I will not attempt to answer*" (developer from mbostock/d3). A second developer states that "*the truck factor is mostly concerned with institutional memory getting lost. No automatic system can account for this lost, unless all project communication is public.*" (developer from libgdx/libgdx). Our survey also reveals that developers usually consider documentation as the best practice to overcome a truck factor episode.

Despite the challenges in computing truck factors automatically, our code-authorship-coverage heuristic presents compelling results. We receive positive or partially positive answers for 30 systems (53% of the valid answers). Even when developers do not agree with our estimation, it is not completely safe to discard a possible damage to the system. For example, six developers state that truck factor is not a concern in open source systems, since it is always possible to recruit new core developers from their large base of contributors. In fact, we observe a successful transition of core developers in at least two systems. In contrast, we cannot discard the risks inherent to such transitions, specially when they should take place due to a sudden and unplanned truck-factor-like episode.

Developers also point two concrete problems in our heuristic for computing truck factors. First, it considers all files in a system as equally important in terms of the features they implement. However, not all requirements and features are equally critical to a system survivability. We address this problem by discarding some files from our analysis, in the cases they lead to highly skewed results (*e.g.*, recipes from Homebrew/homebrew). However, in other systems this partition between core and non-core files is less clear (at least, to non-experts). Second, the heuristic does not account the last time a file is changed. In the survey, some developers claim that losing the author of a very stable file is not a concern (since they probably will not depend again on this author to maintain the file). When such files are common in a system, the heuristic can be adapted to just consider recently changed files.

VIII. Threats to Validity

Construct Validity. We compute the degree-of-authorship using weights derived for other systems [10], [11]. Therefore, we cannot guarantee these weights as the most accurate ones for assessing authorship on GitHub projects. However, the authors of the DOA formula show that the proposed weights are robust enough to be used with other systems, without computing a new regression. Still, we mitigate this threat by initially inspecting the DOA results for 162 pairs of authors and files. Contrasting results with those from `git-blame` suggest DOA-values to be reliable.

The presence of non-source code files, third party libraries, and developers aliases can also impact our results. To address these threats our tool performs file cleaning and alias handling steps before calculating truck factor estimates.

Internal Validity. Our approach computes the authors of a file by considering all the changes performed in the target file. Therefore, our approach is sensitive to loss of part of the development history as result of a erroneous migration to GitHub. We mitigate this threat using a heuristic to detect systems with clear evidence that most of its development history was performed using another version control platform and that this history could not be correctly migrated to GitHub. Moreover, the full development history of a file can be lost in case of renaming operations, copy or file split (*e.g.*, as result of a refactoring operation like *extract class* [20]). We address the former problem using Git facilities (*e.g.*, `git log --find-renames`). However, we acknowledge the need for further empirical investigation to assess the true impact of the other cases.

External Validity. We carefully select a large number of real-world systems coming from six programming languages to validate our approach. Despite these observations, our findings—as usual in empirical software engineering—cannot be directly generalized to other systems, mainly closed-source ones. Many others aspects of the development environment, like contribution policies, automatic refactoring, and development process may impact the truck factor results and it is not the goal of this study to address all of them.

Finally, to assess the impact of the aforementioned threats in our results, we conduct a survey with developers of the systems under analysis in this paper, as reported in Section VI.

IX. Related Work

Although widely discussed among *eXtreme Programming* (XP) practitioners, there are few studies providing and validating truck factor measures for a large number of systems. Zazworka et al. [2] are probably the first to propose a formal definition for TF, specifically to assess a project's conformance to XP practices. For the purpose of simplicity, their definition assumes that all developers who edit a file have knowledge about it. Furthermore, they only compute the TF for five small projects written by students. Ricca et al. [3], [4] use Zazworka's definition to compute truck factors for opensource projects. In their first work, they propose the use of the TF algorithm as strategy to identify "heroes" in software development environments. In their second work, the authors point for scalability limitations in Zazworka's algorithm, which only scales to small projects ($\leq$ 30 developers). In our study, 122 out of 133 systems have more than 30 developers (maximum is torvalds/linux, with thousands of developers among non-driver files). Hannebauer and Gruhn [21] further explore the scalability problems of Zazworka's definition, showing that its implementation is NP-hard. Cosentino et al. [22] propose a tool to calculate TF for Git-based repositories. They use a hierarchical strategy, aggregating file-level authorship results to modules and, in a second step, aggregating module-level results into systems. They evaluate their tool with four systems developed by members of their research group.

Overall, our study differs from the previous ones in three main points: we use the DOA model to identify the main

authors of a file; we evaluate our approach in a large dataset composed of real-world software from six programming languages; we validate our results with expert developers.

X. Conclusion

This paper proposes and evaluates a heuristic-based approach to estimate a system's truck factor, a concept to assess knowledge concentration among team members. We show that 87 systems (65%) have $TF \leq 2$. We validate our results with the developers of 67 systems. In 84% of the valid answers, respondents agree or partially agree that the TF's authors are the main authors of their systems; in 53% we receive a positive or partially positive answer regarding the estimated truck factors.

According to the surveyed developers, documentation is the most effective development practice to overcome a truck factor event, followed by the existence of an active community and automatic tests. We also comment on the main lessons we learn from the developers' answers to our questions.

As future work, we plan to introduce and evaluate the improvements suggested by the surveyed developers in our heuristic to compute truck factors. As an example, we can consider only recently changed files in the estimation of TFs and compute authorship at line level. Finally, we intend to perform a second study, considering industrial and closed systems, and compare the truck factor results with the ones we report in this paper for opensource systems.

Acknowledgments

We thank all the respondents of our survey. This study is supported by grants from FAPEMIG, CNPq, and UFPI.

References

[1] L. Williams and R. Kessler, *Pair Programming Illuminated*. Addison Wesley, 2003.

[2] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider, "Are developers complying with the process: an xp study," in *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 14:1–14:10.

[3] F. Ricca and A. Marchetto, "Are heroes common in FLOSS projects?" in *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 1–4.

[4] F. Ricca, A. Marchetto, and M. Torchiano, "On the difficulty of computing the truck factor," in *Product-Focused Software Process Improvement*. Springer, 2011, vol. 6759, pp. 337–351.

[5] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[6] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006, pp. 361–370.

[7] S. Minto and G. C. Murphy, "Recommending emergent teams," in *4th Workshop on Mining Software Repositories (MSR)*, 2007, pp. 5–5.

[8] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *5th International Working Conference on Mining Software Repositories (MSR)*, 2008, pp. 121–124.

[9] L. Hattori and M. Lanza, "Mining the history of synchronous changes to refine code ownership," in *6th International Working Conference on Mining Software Repositories (MSR)*, 2009, pp. 141–150.

[10] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 385–394.

[11] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: modeling a developer's knowledge of code," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 2, pp. 14:1–14:42, 2014.

[12] K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "Revisiting the applicability of the pareto principle to core development teams in open source software projects," in *14th International Workshop on Principles of Software Evolution (IWPSE 2015)*, 2015, pp. 46–55.

[13] G. Gousios, M. Pinzger, and A. V. Deursen, "An exploratory study of the pull-based software development model," in *36th International conference on Software engineering (ICSE)*, 2014, pp. 345–355.

[14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *11th Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 92–101.

[15] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *22nd International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 155–165.

[16] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, "Feature scattering in the large: a longitudinal study of Linux kernel device drivers," in *14th International Conference on Modularity*, 2015, pp. 81–92.

[17] S. Chacon and B. Straub, *Pro Git*, 2nd ed., ser. Expert's voice in software development. Apress, 2014.

[18] F. Shull, J. Singer, and D. I. Sjøberg, *Guide to Advanced Empirical Software Engineering*. Springer, 2007.

[19] A. Strauss and J. M. Corbin, *Basics of Qualitative Research*. SAGE, 1998.

[20] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[21] C. Hannebauer and V. Gruhn, "Algorithmic complexity of the truck factor calculation," in *Product-Focused Software Process Improvement*. Springer, 2014, vol. 8892, pp. 119–133.

[22] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Assessing the bus factor of Git repositories," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 499–503.