Reviving 86 out of 137 of Tools from ICSE and FSE

Emerson Murphy-Hill
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
emerson@csc.ncsu.edu

***Abstract*—Many innovative software engineering tools appear at the field's premier venues, the International Software Engineering Conference (ICSE) and the Foundations of Software Engineering (FSE). But what happens to these tools after they are presented? In this paper, we describe a course project where we spent thousands of person hours trying to obtain, download, use, and repackage 137 tools from ICSE and FSE's tool demonstration tracks from 2011 through 2014. Our results enumerate the practical and accidental reasons that software engineering tools fail to work over time, and provide practical implications for creating lasting research tools.**

***Keywords*-software engineering tools; replication**

ADDITIONAL AUTHORS: Shabbir Abdul (sabdul@ncsu.edu), Varun Aettapu (vaettap@ncsu.edu), Sumeet Agarwal (sagarwa6@ncsu.edu), Sindhu Anangur Vairavel (sanangu@ncsu.edu), Rishi Avinash Anne (raanne@ncsu.edu), Haris Mahmood Ansari (hmansari@ncsu.edu), Ankit Bhandari (abhanda3@ncsu.edu), Anand Bhanu (bhanua@ncsu.edu), Aditya Vinayak Bhise (avbhise@ncsu.edu), Saikrishna Teja Bobba (sbobba3@ncsu.edu), Vineela Boddula (vboddul@ncsu.edu), Venkata Krishna Sailesh Bommisetti (vbommis@ncsu.edu), Dwayne Christian (Chris) Brown (dcbrow10@ncsu.edu), Peter Morgan Chen (pmchen@ncsu.edu), Yi Chun (Yi-Chun) Chen (ychen74@ncsu.edu), Nikhil Chinthapallee (nchinth@ncsu.edu), Karan Singh Dagar (kdagar@ncsu.edu), Joseph Decker (jdecker@ncsu.edu), Pankti Rakeshkumar Desai (prdesai2@ncsu.edu), Jayant Dhawan (jdhawan2@ncsu.edu), Yihuan Dong (ydong2@ncsu.edu), Sarah Elizabeth Elder (seelder@ncsu.edu), Shrenuj Gunvant Gandhi (sgandhi4@ncsu.edu), Jennifer Michelle Green (jmgree17@ncsu.edu), Mohammed Hasibul Hassan (mhhassan@ncsu.edu), Satish Inampudi (sinampu@ncsu.edu), Pragyan Paramita Jena (ppjena@ncsu.edu), Bhargav Rahul (Bhargav) Jhaveri (bjhaver@ncsu.edu), Apoorv Vijay Joshi (avjoshi@ncsu.edu), Nikhil Josyabhatla (njosyab@ncsu.edu), Sujith Katakam (skataka@ncsu.edu), Juzer Husainy Khambaty (jhkhamba@ncsu.edu), Aneesh Arvind Kher (aakher@ncsu.edu), Craig Kimpel (ckimpal@ncsu.edu), Siddhartha Kollipara (skollip@ncsu.edu), Asish Prabhakar Kottala (akottal@ncsu.edu), Abishek Kumar (akumar21@ncsu.edu), Harini Reddy Kumbum (hkumbum@ncsu.edu), Nitish Pradeep Limaye (nplimaye@ncsu.edu), Apoorv Mahajan (amahaja3@ncsu.edu), Sai Sindhur Malleni (smallen3@ncsu.edu), Sudha Manchukonda (smanchu@ncsu.edu), Kavit Maral Mehta (kmmehta@ncsu.edu), Justin Alan Middleton (jamiddl2@ncsu.edu), Ramakant Moka (rmoka@ncsu.edu), Eesha Gopalakrishna Mulky (egmulky@ncsu.edu), Gauri Naik (gnaik2@ncsu.edu), Shraddha Anil Naik (sanaik2@ncsu.edu), Yashwanth Nallabothu (ynallab@ncsu.edu), Yogesh Nandakumar (ynandak@ncsu.edu), Kairav Sai Padarthy (kspadart@ncsu.edu), Pulkesh Kumar Yadav Pannalal (ppannal@ncsu.edu), Sattwik Pati (spati2@ncsu.edu), Kahan Prabhu (kprabhu@ncsu.edu), Shashank Goud Pulimamidi (spulima@ncsu.edu), Gargi Sandeep Rajadhyaksha (gsrajadh@ncsu.edu), Priyadarshini Rajagopal (prajago4@ncsu.edu), Venkatesh Sambandamoorthy (vsamban@ncsu.edu), Mohan Sammeta (msammet@ncsu.edu), Shaown Sarker (ssarker@ncsu.edu), Anshita Sayal (asayal@ncsu.edu), Vrushti Kamleshkumar Shah (vkshah@ncsu.edu), Esha Sharma (esharma2@ncsu.edu), Saurav Shekhar (sshekha3@ncsu.edu), Sarthak Prabhakar Shetty (spshetty@ncsu.edu), Manish Ramashankar Singh (mrsingh@ncsu.edu), Ankush Kumar Singh (asingh21@ncsu.edu), Vinay Kumar Suryadevara (vksuryad@ncsu.edu), Sumit Kumar Tomer (sktomer@ncsu.edu), Akriti Tripathi (atripat4@ncsu.edu), Jennifer Tsan (jtsan@ncsu.edu), Vivekananda Vakkalanka (vvakkal@ncsu.edu), Alexander Valkovsky (avalkov@ncsu.edu), Rishi Kumar Vardhineni (rkvardhi@ncsu.edu) and Manav Verma (mverma4@ncsu.edu

I. INTRODUCTION

Software engineering research seeks to better understand how software is built and constructed. While such understanding in isolation can be insightful, ultimately the bulk of such research aims to impact practice. To do so, researchers can create recommendations for new software engineering practices, can create educational techniques and materials, and can create tools.

Arguably creating new tools is the most common way that sofware engineering researchers attempt to influence practice. Broadly speaking, tools are software that can help design and build software. Examples include a tool that helps mobile application developers choose which devices to target [12], a tool that checks the use of locks in multithreaded programs [8], and a tool that creates code from natural language text [7].

While papers describe tools in software engineering venues, there are several reasons why software engineering researchers should make the tools themselves available. First, the full details of how the tool works, both in terms of its internals and from an end-user perspective, may not be clear from the paper. Second, making the tools available can help pracitioners try and adopt the tools in their work. Third, it helps the advancement of the field by allowing others to build on existing tools, rather than re-building them from scratch. Finally, it helps facilitate replicability by enabling future researchers to perform studies with the original tools. In this paper, we use the term replicability [1] to refer to "an independent group [obtaining] the same result using the author's own artifacts" [31]. Although artifacts in replications also commonly includes data, in this paper we focus on software.

Despite the benefits of making tools available, in this paper we catalog the practical difficulties in doing so. We describe a study that examined the tools presented at the premier venues for software engineering research over the past several years. As part of a graduate university course on software engineering, we spent 6706 hours trying to obtain, download, use and repackage tools from the International Conference on Software Engineering (ICSE) and the Symposium on the Foundations of Software Engineering (FSE). In doing so, we make the following three main contributions in this paper:

- A study that evaluates the difficulty in getting 137 tools working across a variety of software engineering research.
- A synergistic course project for graduate software engineering students that gives students meaningful educational value *and* provides the research community value.
- 49 existing tools repackaged in automatically-built virtual machines to make it easier for others to use these tools.

II. Related Work

A. Replicability in Science Policy

Our study is one specific foray into the topic of replicability in science, an area whose importance researchers and the public increasingly cite. Ince and colleagues "argue that, with some exceptions, anything less than the release of source programs is intolerable for results that depend on computation" [9].

Both in science generally and in computer science specifically, the urgency of increasing replicability has increased sharply. In a 2016 survey of 1,500 scientists across disciplines, the journal Nature reported that 70% of researchers have attempted and failed to replicate others' experiments [2]. A 2015 study in the journal Science attempted to replicate 100 psychology experiments, finding that a "large portion of replications produced weaker evidence for the original findings" [4].

In computer science, the Association for Computing Machinery has recently formed the "Task Force on Data, Software, and Reproducibility in Publication" and the "Replication Taskforce." The goals of these taskforces are to "promote practices that lead to better replicability" [28] and "develop proposals on how ACM can bring current replication and verification practices in line with the rest of the scientific community" [20]. While both task forces are actively working as of this writing, in 2016 they released a policy that recommends certification of replicability and reproducibility [31]. But because few replication studies exist in computer science, the motivation for this policy is based on studies "primarily in the biomedical field." Our study provides data to inform such evolving policies.

B. Replicability Studies in Computer Science

Some computer science subdisciplines have attempted to replicate research involving software. Kovačević evaluated 15 image processing papers, finding that algorithm descriptions were generally adequate, but that none of the papers provided implementations [11]. Later, Vanderwalle and Kovačević repeated that study on 134 image processing papers and found that 84% provided implementation details and 9% provided code [19]. Our study differs in that we went beyond finding out whether code was available; we ran and repackaged that code.

Callberg and Proebsting's recent replicability study [6], [5], published after we had completed our study, is most similar to ours. In it, the authors examined the systems described in 601 papers from 13 systems conferences from 2011 to 2013 (including, relevant to our paper, OOPSLA and PLDI 2012). Of those papers:

- in 85 cases, the authors provided code in the article;
- in 54 cases, code could be found via web search;
- in 87 cases, the authors provided code upon request;
- in 146 cases, authors refused to provide the code; and
- in 30 cases, authors did not respond to email.

Furthermore, Callberg and Proebsting's team tried to get the tools to build, finding that:

- in 130 cases, the code was built in 30 minutes or less;
- in 64 cases, the code was built in more than 60 minutes;
- in 23 cases, the code's original authors assured the team that the code could be built; and
- in 9 cases, the code could not be built.

Apart from studying tool demonstrations from software engineering venues rather than technical papers from systems venues, our study primarily complements Callberg and Proebsting's study in two ways. First, we used a stricter standard of acceptance; rather than just building the tool, we verified that the tool ran as described in the original paper. Second, in addition to validating that the tool works, we attempted to repackage the tools as a contribution to the community, so that other researchers would not have to duplicate our efforts.

Other papers in software engineering have also explored replicability with tools. Klein and colleagues selected 9 papers from ICFP 2009 to re-implement as formal models in their domain specific language; they found mistakes in all 9 papers [10]. Mende attempted to replicate prior results using two existing defect predictors, finding that even "with access to the original data sets, replicating previous studies may not lead to the exact same results" [13]. More recently, Shepperd and colleagues' replicated 42 studies using defect predictors, finding that the research group performing the analysis has a strong effect on performance [17]. Likewise, Tantithamthavorn and colleagues' replication of this meta-study confirmed these results, but explained the problem as arising from research groups reusing datasets and metrics [18]. Our study complements these papers in that we evaluate a wider variety of software engineering tools.

C. Artifact Evaluation Committees

Researchers in programming languages, and to a lesser extent software engineering, have recently implemented steps to increase reproducibility by forming *artifact evaluation committees* (AECs). AECs run in parallel with program committees, and aim to review research artifacts, including tools. At the top sofware engineering conferences, AECs have been run at the main technical track of Foundations of Software Engineering 2011 and 2013 through 2015. Accepted research papers are encouraged, but not required, to submit artifacts to AECs. Accepted artifacts may or may not be archived by the authors after the conference.

At least 10 past instantiations of AECs have reported data about their outcomes, which Collberg and colleagues summarize [5]. They find that typically about half of authors of accepted papers choose to submit artifacts, and that acceptance rates for those artifacts vary substantially.

The results of AECs and our study have similar goals and technical approaches. Both aim to evaluate the artifact's consistency with the paper, completeness, documentation, and ease of use [23]. From a technical standpoint, AECs generally encourage packaging of artifacts in virtual machines.

However, AECs and our study are different in several ways. AECs sometimes evaluate non-tools, such as data analysis scripts; our study evaluates only software engineering tools. AECs report only data from artifacts that authors choose to submit; we report data about all tools published at a conference's tool demonstration track. AECs evaluate artifacts packaged by the authors themselves; we report on tools that we package ourselves as outsiders. AECs aim to evaluate artifacts during publication; our study aims to evaluate tools years after publication. AECs do not require the public distribution of artifacts; distribution of artifacts is one of our primary goals.

Regardless of the similarities and differences, our study motivates the need for AECs by demonstrating the the situations in which tools "disappear" years after publication. At the same time, our study highlights some significant challenges to AECs by demonstrating the several fundamental reasons tools cannot be archived.

III. Research Questions

In this paper, we will simply say *tool* to refer to software engineering tools presented at the International Conference on Software Engineering or Foundations of Software Engineering in their respective tool demonstration tracks. We will also use the term *researchers* to refer to the creators of the tool described in the paper.

1) How much effort is required to get tools to work?
2) What are the barriers to get tools to work?
3) What are the barriers to get tools to work in virtual machines?
4) To what extent is this study practical to implement in a classroom context?

IV. Course Description

We conducted the study described in this paper as part of a graduate software engineering course in the Computer Science department at North Carolina State University. The department has about 200 PhD students and 450 MS students in its graduate program. While graduate students are not required to take the course, it is one of seven core systems courses, from which students must take one course. The department does have a specialty MS degree track in software engineering [26], which does require this course.

Course content covers software engineering processes, software architecture, design patterns, software security, verification and validation, estimation, project management, requirements, certification, and formal methods [21]. Apart from the course content, involving lectures and a final exam, the other major component of the course is the project. Next, we describe the project for this course in prior offerings of the course (Section IV-A) and in the offering described in this paper (Section IV-B).

A. Prior Course Project and Criticism

In prior offerings of the course, students were essentially given two project offerings. In one version, students could create a software engineering tool, such as a plugin for Eclipse. In the other version, students chose several existing, similar software engineering tools, and applied them to open source software, then reflected on what they learned about the tools and the projects. Most students opted for the second version. In both cases, students were required to write up their results. This project is similar to a course project assigned by David Notkin at the University of Washington [22].

After offering this project to students for several years, the instructor recognized several problems the way he had executed it:

- The course did not have enough time to teach students technical writing skills, and thus final papers were of poor quality, on average. Moreover, while technical communication is a course objective, other types of communication would likely be more valuable to most students, who are non-thesis and industry-focused.
- Students had limited exposure to state-of-the-art tools; the tools they chose to study were typically quite basic.
- Students appeared to rarely gain technical skills during the project.
- Most students were not doing *novel* work; each semester, different students wrote similar reports on similar tools, which had no value beyond their educational value.

While the last point may seem odd – why would a course project need value beyond its educational value? – it's not unusual for some software engineering courses to have added value, such as contributing to open source projects [16], [14]. In the case of the present course, the instructor thought that such system development would not be useful to most students in the course, who typically come in with 1 to 2 years of industrial software engineering experience.

B. New Course Project

In the Fall of 2015, the instructor changed the course project to alleviate the problems described in the last subsection. In short, student teams were assigned tools described in a a prior research paper at ICSE or FSE, the premier venues for software engineering research. Students were required to obtain the tools, get them running, and redistribute them.

The instructor chose tool demonstration papers, rather than full techncial papers, for practical reasons. Full technical papers may not present a tool; for instance, a purely qualitative study that reports on empirical findings may have no software to go along with it. In contrast, tool demo papers almost certainly had a working tool when the paper was presented at a conference.

There were several learning goals of the course project:

- Gain deep experience with several state-of-the-art software enginering tools, and broad overview of many others;
- Effectively read research papers;
- How to build virtual machines that contain custom software;
- How to script virtual machine creation; and
- Oral communication skills.

In the remainder of this section, we describe project activities, requirements, and deliverables.

1) Team Formation and Tool Selection: The instructor formed teams of about 5 students by randomly selecting about 4 on-campus students and possibly 1 off-campus, distance education student.

During class, the instructor asked teams to find the ICSE and FSE demonstration tracks online, skim the papers from those tracks, and extract several pieces of information: the venue, the paper name, the tool name, any links to source code or binaries for the tools, the technologies involved in the tool's creation, and a rough estimate of how difficult the team thought it would be to get the tool working. Teams put this information on a shared spreadsheet with one row per paper. We collected papers from from 2014, the most recent year either ICSE or FSE papers will officially posted online, to 2009. In total, we collected 188 papers from 12 conferences.

Teams were then given the opportunity to look over the list of tools, and identify tools they might want to work on. During the following class period, teams chose N tools to work on, where N was the number of people on the team. Each tool could be assigned to one and only one team.

Because some tools were in high demand and some in low demand, for fairness, teams chose tools in a round-robin draft, that is, one team chose a tool, then the next team chose a tool, and so on, until all teams had chosen one tool, and then the first team chooses a second tool, and the next team chooses a second tool, and so on. At the time of the draft, 100 students were present in the class, so the tools eligible for selection were the 100 most recent tools, which included all tools from 2012–2014, and a few from 2011.

2) Obtaining Tools: After teams were assigned tools, their first task was to obtain the tools, including the source code, if possible, and the binary if not. For tools that were deployed (or partially deployed) via a web service, students were required to obtain the source or binary for the service as well; in short, the students needed to obtain all software components necessary to run the tool in isolation.

Teams were instructed to first try to obtain the tool from links in the paper itself or via the web. If teams could not find the tool, the teams were instructed to ask the researchers, using an email template show in Figure 1. To minimize communication with the researchers, the email contained requests

To: <AUTHORS OF PAPER, BUT NOT MORE THAN THE FIRST 5 AUTHORS>

Subject: Regarding your tool, <TOOLNAME>

Dear Dr. <LASTNAME> and colleagues,

I enjoyed reading your paper <PAPER TITLE>. As part of a graduate software engineering class at NC State University, my team has chosen to use the tool <TOOLNAME> as part of our class project. In short, the class is using tools from the past few ICSEs and FSEs, then putting all those tools in an accessible form (e.g., virtual machine) in central location. Our project is supervised by Dr. Emerson Murphy-Hill (CC'd via ncsu-csc-510@googlegroups.com). When our project is complete, we plan on aggregating our results (e.g., how many tools could we get working, how easy was it, etc) into a research paper.

Because my grade rests on your tool, I am motivated to get it working. I have a few questions for you before I get started.

May I have permission to redistribute an executable version of your tool? Specifically, we plan on putting your tool in a virtual machine, then posting that VM on the public internet.

May I have permission to redistribute the source code for your tool? Specifically, I plan on getting the tool building into the virtual machine (by way of Vagrant) and posting the build scripts on GitHub.

Could you please send me a link (or attachment) to the executable version of your tool? I have attempted to find the tool on the internet, but have been unsuccessful.

Could you please send me a link (or attachment) to the source code of your tool? I have attempted to find it on the internet, but have been unsuccessful.

I've had some trouble getting the tool to work... <describe what you tried, describe what you expected to happen, describe what actually happened.>

Thank you very much,

<YOUR NAME>

Fig. 1. An email template used by students for obtaining information about tools from researchers.

for several pieces of information, the need for which will become clear later in this section. Students were instructed to customize this template, but to use it as a starting point. Full instructions for use of this template can be found online [27]. Teams were asked to include the instructor in all communications with researchers.

3) Getting the Tools Working: Once students had obtained the tools, they had two weeks to get the tools working. Because some tools may only work in the very specific situations described in their papers, teams were required to get the tools "working" as it was described in the paper. Teams were allowed to use any means necessary to get the tools working, including asking classmates for help, but were discouraged from pestering the researchers. At the end of the two weeks, if the tool was obtained and working, teams certified that the tool as working on the spreadsheet.

If a team could not get a tool working for whatever reason, several things happened. First, the team must certify the tool as "unworkable." Second, the team would be assigned a new tool by moving down the list of tools, and the process would start again. Third, the tool certified as "unworkable" would be made available to the other teams to try to get working.

The instructor created a disincentive for students to unnecessariyl certify a tool as unworkable. If a team got an unworkable tool working, the team that certified it as unworkable had their final course grade reduced by a minor grade (for example, from an A to an A- or from a C to a C-). The team that gets the "unworkable" tool working gets their final grade increased by a minor grade point. Furthermore, the only way to get an A+ course is to be on a team that gets an unworkable tool working.

4) Getting the Tools Working in a Virtual Machine: The first graded deliverable was a VirtualBox virtual machine image that contained an operating system, the tool, any and other software required by the tool (such as Eclipse), documentation, and license information. The goal of creating this image was to make it as easy as possible for future potential users to try the tool out; in essence, any "fiddling" required to get a tool working would not have to be done by the user.

The grading rubric for the image [29] additionally specified that minimal work is required on the part of the user to see the tool in action, that the technology stack does not contain any proprietary software that is not not stricly necessary, and that the image is as small as possible.

5) Building the Virtual Machine Image Automatically: After building a basic virtual machine image by hand, teams next built Vagrant scripts that built virtual machines automatically. The main goal of creating the script was to add transparency to the process of installing the tool; if future users want to install the tool in their own development environment, the script provides a specification for doing so.

The grading rubric for the virtual machine script [30] was largely the same as for the hand-built image. For instance, some things that were easy to do in a hand-built image turned out difficult in the script, such as changing the username and password, so such requirements were not included in the grading rubric for the script. The most substantial additional requirement for the script was that it uses "standard and stable external resources whenever possible (e.g., www.vagrantbox.es, rather than a hand-built box)".

6) Redistributing the Tools: Teams were required to establish GitHub repositories for each tool that their team was assigned – working or not. The main goal was to share the work the teams had done with others. We created an organization to house each repository, which can be found online.[1]

If the original tool contained a license to redistribute the source code, or the researchers gave us explicit permission to do so, we redistributed that code in our tool's repository. The same applied to the tools' binary. If the tool was working, vagrant scripts were also included in the repository, regardless of whether we had permission to redistribute the tool; if we did not, users would have to contact the researchers and drop in the tool (a binary, for instance) for the script to work. Even if the tool was not available to us, we created a mostly empty repository, for consistency.

For tools for which we had source code, whenever possible teams included the history of that source code, for completeness. Ideally, tools that were already hosted on GitHub could be forked by the team, maintaining not just history but also an explicit connection to the original repository. When original tool were hosted elsewhere, such as in subversion on Google Projects, teams migrated source code history to GitHub.

Teams added a readme file to each tool's repository that conveyed some basic information about the tool, including links to the original paper and original project webpage, as well as acknowledgements. The rubric outlined a number of small, specific requirements for the readme to ensure consistency [24].

When teams had permission to redistribute the virtual machine image containing the tool, readmes also contained links to those images. The images were hosted on the instructor's Google Drive account, which provides unlimited cloud storage.[2] Such large capacity storage is necessary because each image occupies several gigabytes of space.

7) Presenting Tools: For the communication part of the class, teams were to present each working tool in front of the class for a 5 minute demonstration. The grading rubric specified, among other requirements, that the presentations explained what problem the tool was built to solve, to give a simple enough example that the tool could be understood, and to be realistic enough to be compelling [25]. Teams also prepared a video demo of each working tool, posted on YouTube and linked to from the virtual machine image and the GitHub readme.

8) Evaluations: Teams' deliverables were evaluated by their peers and by two teachers' assistants (TAs). While both peers and TAs evaluated the deliverables for quality and consistency, only the TA evaluations counted towards teams' grades.

Teams completed peer evaluations of other teams' hand-built virtual machines and GitHub repositories. This entailed reading the original paper to understand how the tool was supposed to work, using the tool in the virtual machine, and comparing deliverables to the grading rubrics. When students found defects, they filed issues in the repositories issue tracker and provided fixes for simple defects using pull requests.

Hand-build virtual machine images were evaluated in two rounds by the TAs, where students enahanced their images between rounds and the rubrics were updated for the second round based on what the TAs observed in the first. Likewise, vagrant scripts were evaluated in two rounds by the TAs.

[1] https://github.com/SoftwareEngineeringToolDemos/

[2] https://support.google.com/a/answer/2856827

Finally, TAs evaluated all artifacts (images, vagrant scripts, and repositories) together in a final project evaluation.

9) Data Collection and Contributions: As part of the final project submission, the instructor collected data about each tool in the form of a survey. This tool survey asked, for instance, how long the team estimated they spent on getting the tool working. Teams reported data for all tools, except six tools, all from a single team. The team certified six tools as unworkable, but all students who worked on the tools dropped the course. The remaining student was unable to provide accurate data about the tools, and thus no reports were submitted.

Each student also submitted a second survey that asked students what they thought about the project, and whether they wanted to participate as an author of this paper. Two students opted not to participate as an author. Data was also collected from standardized and anonymous course evaluation forms, forms that are used for all courses across the university.

This paper was written primarily by the first author, the instructor of the course, after the course had ended. The paper's source and history can also be found on GitHub.[3] The technical work was conducted primarily by the students.

V. Example Tool

In this section, we exhibit one workng tool. Figure 2 shows our GitHub page for the tool jStar-eclipse [15]. This page shows that we forked the repository from an existing GitHub repository. The README.md has the name of the tool, the contents of the original readme (the first two lines under the title), links to some original resources, and acknowledgements. It also states that the repsoitory contains the source code, a link to the original tool, binaries, and an external link to our virtual machine.

Figure 3 shows an the Vagrant script and part of an external script that builds the virtual machine. The vagrant script installs a plugin, uses a base virtual machine image from the boxcutter community,[4] and calls four shell scripts interspersed with operating system restarts. The external shell script shown removes some unneeded software, and installs some prerequisites to the tool, including Eclipse and an OCaml compiler.

Figure 4 shows the running virtual machine for jStar-eclipse, as downloaded. In the figure, Eclipse has an example project loaded that demonstrates the tool, the tool plugin is installed into Eclipse, Eclipse is installed on Ubuntu linux, and Ubuntu is running in a virtual machine running Windows 10. In short, the user is relieved of having to install the technology stack to run the tool; instead, she only has to download the virtual machine image.

In the remainder of the paper, we do not call out any particular tool, but instead only speak of aggregates. The reason is that we do not wish to embarass any of the researchers. Accordingly, we have taken steps to make sure that

[3]https://github.com/SoftwareEngineeringToolDemos/paper
[4]https://atlas.hashicorp.com/boxcutter

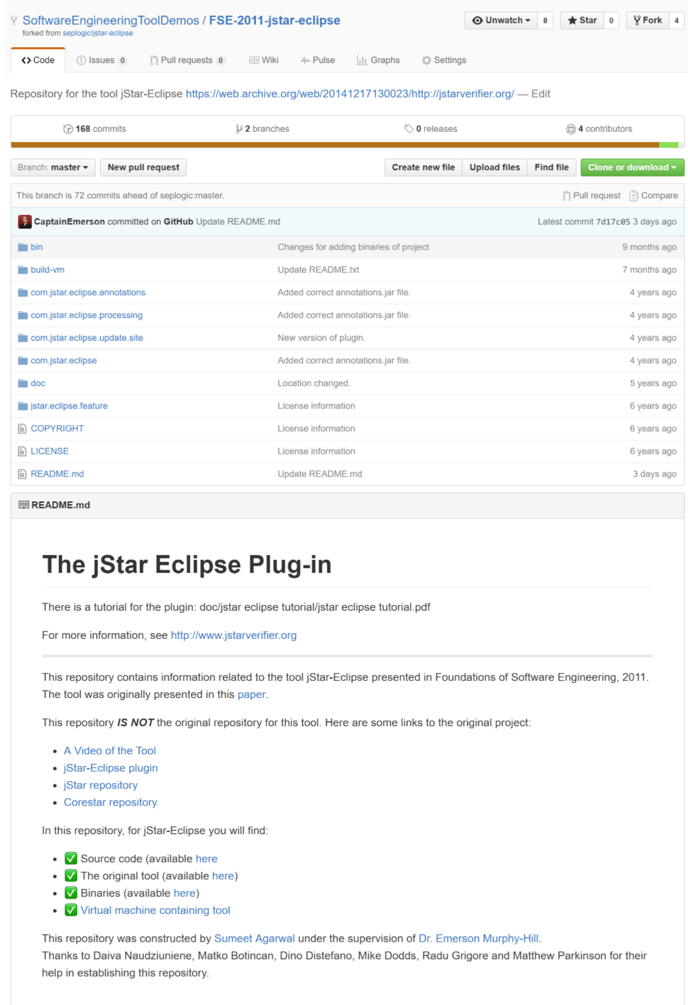

Fig. 2. Our forked repository for a jStar-eclipse.

data we have made public with this paper does not embarass researchers either. For instance, while our GitHub repositories say when tools are not available, we do not publicly indicate why they are not available.

VI. Results

A. RQ1: Effort to Get Tools Working

Through a survey near the end of the course, teams estimated how long they spent corresponding with researchers, getting a tool to work, getting it to work in a virtual machine, and getting it to work with vagrant. The median time teams spent corresponding with researchers was 2 hours. For tools where teams obtained the tool, they spent a median of 10 hours trying to get it to work. For tools where the teams got a tool working, they spent a median of 10 getting it working in a virtual machine image and 15 hours generating a working tool image with Vagrant. In total, teams spent 6706 hours working on all tools combined.

The survey also asked participants if they spent additional time on their projects. In addition to project requirements like

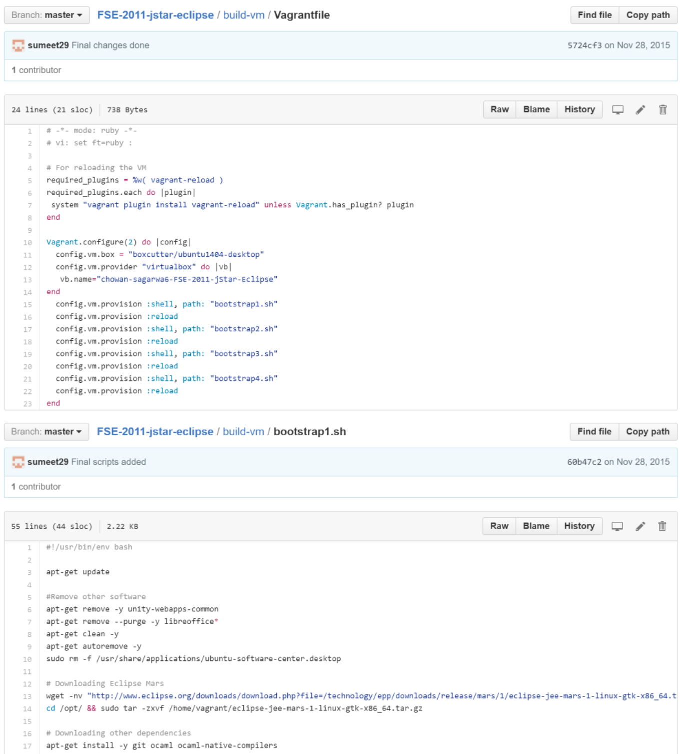

Fig. 3. Scripts for generating virtual machine image containing jStar-eclipse.

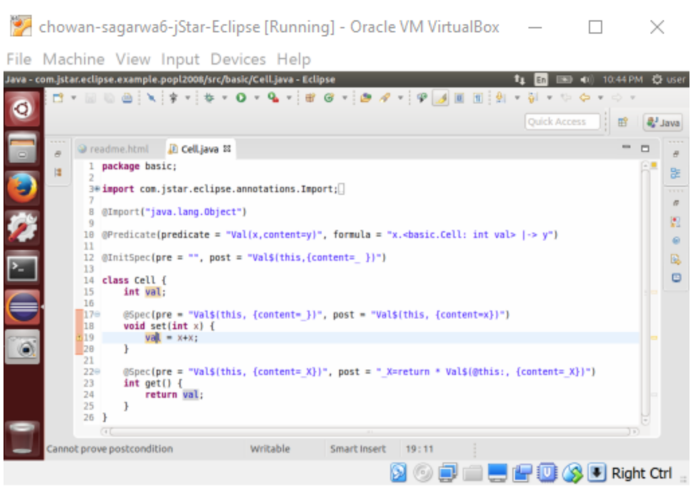

Fig. 4. Screenshot of jStar-eclipse in a virtual machine.

establishing the GitHub repository, teams reported spending additional time learning about their tools, searching for the tool on the web, waiting for researcher replies, obtaining the license for a tool, figuring out what dependencies a tool had, obtaining a Solaris base box for Vagrant, and getting a tool to work with a variety of examples.

Because a significant portion of participants' time was spent corresponding with the researchers, the survey also asked teams about their email interactions. Overall, teams sent emails for 92% of tools. Teams sent a median of 3 and recieved a median of 2 emails from each researcher. Figure 5 summarizes

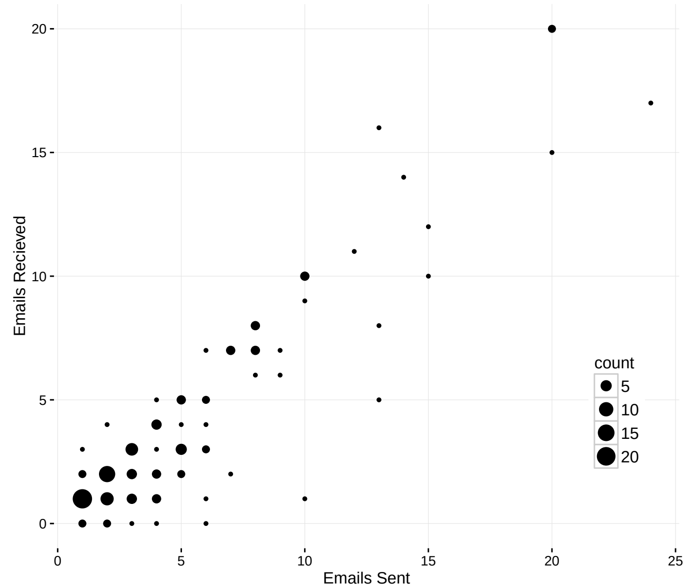

Fig. 5. Summary of emails sent and received.

the number of emails we sent and received, where the size of each dot represents the number of tools that had that many emails sent and received. We see that, overall, few emails were required and researchers appeared generally responsive.

The effort spent to get a tool working was not only borne by us, but also the researchers who read and responded to our emails. To minimize researchers' effort, we did not ask researchers directly. Instead, we asked teams to estimate how much time researchers spent reading and responding to our emails, including time for any background work such as restarting servers. Teams estimated that a median of 15 minutes was spent, with 20 researchers having to spend more than an hour.

B. *RQ2: Barriers to Getting Tools Working*

1) Obtaining the Tools from the Paper: Teams found that about 62% papers contained links to online information about their tools. Team also found that 7% papers had links that were no longer functional, where 44% of those links being fixed after teams emailed the researchers.

When it came to obtaining the tools, 42% could be obtained from links in the papers directly. 19% could not be obtained from links in the papers themselves, but could be from searching the web. When those failed, another 20% were obtained after emailing the researchers. In doing so, 3 links to tools were fixed by the tools' researchers. 20% of tools could not be obtained by teams at all.

We also assessed where the source code for tools are available, apart from our GitHub repositories. 39% of tools had no source code available on line. 28% of tools had source code hosted on an researcher's personal website. Many tools were available on open code repositories: 25% on GitHub, 4% on Google Code, 3% on Souceforge, 3% on Bitbucket, and 2% on CodePlex. Of these, 4 tools began hosting on a public repository after we contacted the researchers.

Even when tools were ostensibly available, sometimes critical pieces were missing, as reported by teams in freeform text on the survey. Teams reported missing configuration files settings that needed to be edited before use, information about IDE compatibility, sample input, and steps required for installation.

2) Researcher Compliance: While researchers were generally responsive to emails, they were not necessarily responsive or compliant with the requests contained in those emails. Figure 6 lists what teams requested from researchers (horizontally) and whether researchers complied (colors). Overall, each kind of request was met with substantial non-response and non-compliance. Beyond these requests, teams reported needing to ask for license keys and an old version of a dependency.

For 13 tools, teams reported asking for a tool, but recieving no response from the researchers. For 3 tools, teams eventually got a response with the tool, but it was too late (more than 7 days after initial contact). 9 tools were not available outside the organization they were developed in. 2 tools had software dependencies on for-pay tools; the may have worked if we had been willing to pay for the dependencies.

Several tools were not available to us, and the researchers were unwilling or unable to do so. For two tools, the researcher was simply unavailable, without further explanation. For one tool, the researcher was uncomfortable making the tool available. For another, the researcher said the tool was only a prototype. For another, the researcher stated that the tool had too many dependencies with other tools. For another, the researcher stated that the tool does not exist anymore.

Teams also reported about the tone of the response they received from researchers. Teams felt that 82% of responses were helpful and 85% were friendly. Only 4% felt the responses were intimidating and 10% seemed annoyed.

3) Technical Challenges: In total, teams marked 63% tools as working. The remaing tools were marked "unworkable" for a several of reasons. The most common reason teams marked tools as unworkable was that the teams could not get the tools working (19 tools), typically due to build errors or mismatches between the way the downloaded tool worked and the way it was described in the paper. A common problem for these tools was dependencies on software whose versions were unspecified (and at least once forgotten by the researcher), and teams failed to get the tools working with the versions they tried. Teams also described having issues with tool configuration, missing functionality, and missing or corrupted program files or data. After contacting the researchers, teams reported that researchers fixed bugs in 8 tools, fixed deployment issues for 2 tools, and fixed some problems with dependencies.

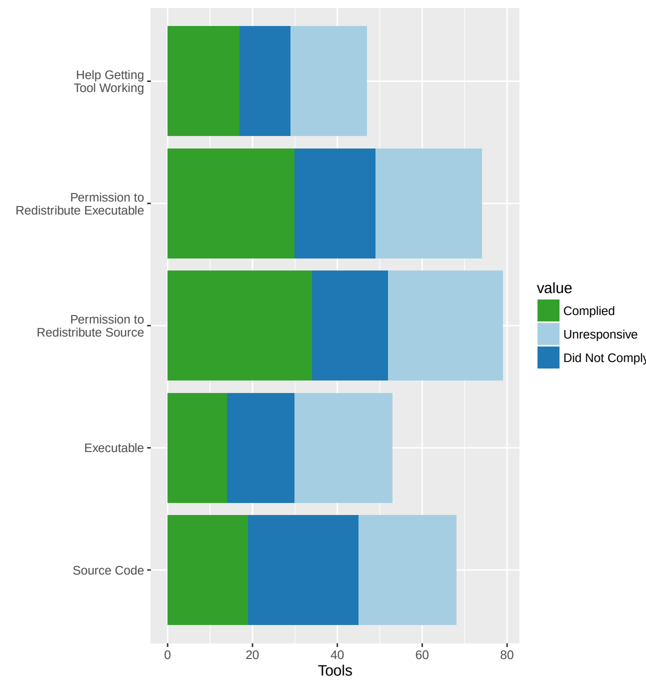

Fig. 6. Researchers compliance with teams' tool requests.

C. RQ3: Barriers to Virtualizing

Once the tools were available and working, the barriers to getting them working in the virtual machines were largely about licensing issues surrounding each tool and the software it depends on.

43 tools were open-sourced, under a variety of licenses, making them appropriate for redistirubtion in virtual machines. Common licenses were: 14 had an Eclipse Public License license, 7 had an MIT license, 7 had an Apache 2.0 license, 6 has a BSD 2.0 license, 5 had a GPL 2.0 license, 4 had a GPL 3.0 license, 2 had a LGPL 2.0 license, 2 had a Microsoft Public License. Researchers added open source licenses to 2 tools after we contacted them by email. Although they did not have an explicit license, researchers of 31 tools granted us permission to redistribute via email. Two tools were freely available online with an open source license, but the researchers specifically asked us *not* to redistribute the tool; although we honored the researchers' requests, the requests nonetheless directly conflicts with those tools' licenses.

Our ability to redistribute virtual machines of the tools were further restricted by the tools' dependencies. The most common dependencies were on the Windows operating system (22 tools). 6 tools were dependent on Visual Studio. A few other tools depended on other commercial operating systems and tools.

After excluding tools that we did not get working, that we did not have permission to redistribute, and that relied on software we did not have permission to redistribute, 49 tools remained. We have made these tools publicly available in virtual machines images online.[5] Our GitHub repositories link to these virtual machine images, and contain relevant tool artifacts such as source code and binaries. A total of 62 repositories contain vagrant scripts to build virtual machines that contain the tools.

D. RQ4: Classroom

Viewed in the context of a university course, students viewed the course project negatively, based on two main metrics. First, many students dropped the course; at its peak, the course was full and had a full waiting list with a combined 105 students. By the end of the course, only 58 students remained. Second, the university-administered evaluations for the course and course project were low. All quantitative metrics for the course were below the Computer Science department average, and were lower than for courses the instructor has previously taught. Specifically, the most common response to the question of whether the course project was a valuable aid to learning was "strongly disagree". What accounts for this negative reaction?

With respect to students dropping the course, students were asked in the non-anonymous survey at the end of the course whether they knew students who dropped, and if so, whether they knew why. While students cited some external reasons, the primary one appeared to be that the course was more work than they had expected. Although other changes had been made to the course, most notably "flipping" the lectures [3], the main one was the project, so the project load was likely responsible for the drops.

With respect to low student evaluations of the project, the problem appears to be with several aspects of project design, rather than a fundamental issue with the project itself. In the non-anonymous survey, students were given a brief description of the old project, and were asked which version of the course project they would prefer. 45% of students preferred the present course project, 26% would prefer the past course project, and the remaining students had no preference. Nonetheless, several areas of the course project were challenging, as described below.

Lack of Teamwork. Teamwork was critical to a successful project, because a single student is unlikely to have all the technical skills necessary to get an arbitrary tool working. While the instructor explained this to students early on, he did not provide an environment that was conducive to teamwork. First, no specific collaboration technologies or practices were encouraged, such as Slack, scrum, or cloud-based virtual machines. Second, the easiest way to break down work was to assign one person in a team to work on one tool; because each tool generally worked independently from other tools, it appeared that teammates worked in a fairly siloed way. Third, teams lacked cohesiveness because teammates were assigned randomly. Because many students dropped the course, team cohesiveness may have been reduced further. Teamwork problems can likely be improved substantially in the future by educating students on and providing collaboration technologies, and allowing teams to choose their own members.

Uneven Tool Difficulties. Some tools are simple, others are complex; likewise, some tools were easy to get working, while others were extremely difficult. The instructor originally aimed to improve fairness by assigning multiple tools to teams, so that average tool difficulty across teams should be roughly equivalent. However, the siloing within teams still meant that some students were doing significantly more work than other students. Breaking down the silos may improve this inequity problem. This may also be an opportunity to create an assignment that applies the course material on effort and risk estimation.

Vagrant Scripting. Vagrant scripting turned out more challenging than anticipated. One reason was that install software on Windows machines is difficult due to operating system security checks. Another difficulty was some scripts needed large files, which GitHub could not store, so these scripts needed to load data from other locations (such as Google Drive), decreasing the likelihood that these scripts will continue to function correctly over time. Building vagrant vagrant scripts should be easier in the future, now that we have a repository of examples that include, for instance, how to use a Windows package manager and how to install Eclipse and load Eclipse workspaces. One way to approach quality problems is by treating them as testing and continuous integration problems; Travis CI, for instance, may be appropriate for building VMs automatically, assessing the quality of those VMs with test automation, and making public the VMs and Travis scripts.

Unusual Grading. The instructor recognized that the grading strategy of making substantial grade adjustments when teams got unworkable tools working would be controversial among students. Four out of 44 students complained about this in the anonymous course evaluation. Only one team ended up getting an unworkable tool working, and this team expressed some hesitation in doing so because it meant a negative repercussions for their classmates. Nonetheless, in the opinion of the instructor, the grading strategy was effective in making sure teams tried sufficiently hard to get their tools working. Future iterations of the project may explore alternate grading options.

In-Class Presentations. Although in-class presentations were only 5 minutes per tool, as a whole they took up an inordinate amount of class time. For future iterations of the projects, students will instead make videos posted online about their tools, and perhaps only one tool per team will be presented in class.

Finally, despite the challenges with the course, the instructor's opinion is that the course was highly worthwhile. He knows of no other software engineering courses that integrate research into teaching to the degree that this one does, while at the same time capitalizing on the existing talents of computer science graduate students for the greater good. While educational outcomes can be strengthened by deeper integration of the lecture material into project work, as a whole, the instructor believes that the project was an

[5] http://go.ncsu.edu/SE-tool-VMs

improvement over the prior project. On the other hand, the project was so radically different from prior projects – perhaps to a mutinous degree – that trying it is a risky proposition for untenured instructors.

VII. Discussion

What are the implications for researchers? For conferences?

Come back to interesting findings: * Services (appendix) * Disappearing tools * license things, people! * OSS licenses, but please don't redistribute * Email is soooo bad; non-response vs. non-compliance. lots of requests just "got lost" * Should AECs use vagrant scripts? Good for archival purposes. * One researcher said tool just doesn't exist anymore. * Another researcher had to dig tool out of long term archive. * Several researchers (what percent?) has tools hosted on Google code, even though it was dying. Did we save 'em? * Threats: quality, etc (See Shriram's replication to demonstrate problems). Data reliability.

A. Implications for Research

Arguably as SE researchers we should be the best!

From perspective of someone making conference, it's desirable to make it easier to use tools.

What do we require for demos?

How does this extend to main track papers?

Have we really solved the bitrot problem? Consider problem of Topaz (?), system that maybe had circular menus; but what did they look like, and what were they for.

How does this jive with current effort for artifact evaluation committees? Complementary;

B. Implications for Education

C. Implications for Practice

Making it easy to demo SE tools would be good for: * those who want to try a vendor's tools *

Challenges: * like elsewhere, get over operating system problems

VIII. Conclusions

Some other things.

Acknowledgments

Thanks to Tim Menzies, Gail Murphy, and Thomas Zimmermann for their helpful comments on drafts of this paper.

References

[1] International vocabulary of basic and general terms in metrology (vim), 2004. http://www.bipm.org/en/publications/guides/vim.html.

[2] M. Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454, 2016.

[3] J. L. Bishop and M. A. Verleger. The flipped classroom: A survey of the research. In *ASEE National Conference Proceedings, Atlanta, GA*, volume 30, 2013.

[4] O. S. Collaboration et al. Estimating the reproducibility of psychological science. *Science*, 349(6251):aac4716, 2015.

[5] C. Collberg, T. Proebsting, and A. M. Warren. Repeatability and benefaction in computer systems research. Technical report, University of Arizona, 2015. TR 14-04.

[6] C. Collberg and T. A. Proebsting. Repeatability in computer systems research. *Communications of the ACM*, 59(3):62–69, 2016.

[7] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In L. K. Dillon, W. Visser, and L. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 345–356. ACM, 2016.

[8] M. D. Ernst, A. Lovato, D. Macedonio, F. Spoto, and J. Thaine. Locking discipline inference and checking. In *ICSE'16, Proceedings of the 38th International Conference on Software Engineering*, Austin, TX, USA, May 18–20, 2016.

[9] D. C. Ince, L. Hatton, and J. Graham-Cumming. The case for open computer programs. *Nature*, 482(7386):485–488, 2012.

[10] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *ACM SIGPLAN Notices*, volume 47, pages 285–296. ACM, 2012.

[11] J. Kovacevic. How to encourage and publish reproducible research. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–1273. IEEE, 2007.

[12] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, D. Hao, G. Huang, and F. Feng. Prada: prioritizing android devices for apps by mining large-scale usage data. In *Proceedings of the 38th International Conference on Software Engineering*, pages 3–13. ACM, 2016.

[13] T. Mende. Replication of defect prediction studies: problems, pitfalls and recommendations. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 5. ACM, 2010.

[14] A. Meneely, L. Williams, and E. F. Gehringer. Rose: a repository of education-friendly open-source projects. In *ACM SIGCSE Bulletin*, volume 40, pages 7–11. ACM, 2008.

[15] D. Naudziuniene, M. Botincan, D. Distefano, M. Dodds, R. Grigore, and M. J. Parkinson. jstar-eclipse: an ide for automated verification of java programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 428–431. ACM, 2011.

[16] M. Pedroni, T. Bay, M. Oriol, and A. Pedroni. Open source projects in programming courses. *ACM SIGCSE Bulletin*, 39(1):454–458, 2007.

[17] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[18] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Comments on" researcher bias: The use of machine learning in software defect prediction". *Transactions on Software Engineering*, 2016.

[19] P. Vandewalle, J. Kovačević, and M. Vetterli. Reproducible research in signal processing. *Signal Processing Magazine, IEEE*, 26(3):37–47, 2009.

[20] The acm special interest group governing board replication taskforce. Accessed August 2016. https://github.com/sharpic/Replication-Taskforce.

[21] Csc 510 syllabus. Accessed August 2016. https://github.com/SoftwareEngineeringToolDemos/Paper/blob/master/course.syllabus.pdf.

[22] Cse503: Software engineering. Accessed August 2016. https://courses.cs.washington.edu/courses/cse503/.

[23] Experimental evaluation of software and systems in computer science. Accessed August 2016. http://evaluate.inf.usi.ch/artifacts.

[24] Github repository rubric. Accessed August 2016. https://github.com/SoftwareEngineeringToolDemos/Paper/blob/master/github.rubric.xlsx.

[25] Github repository rubric. Accessed August 2016. https://github.com/SoftwareEngineeringToolDemos/Paper/blob/master/presentation.rubric.xlsx.

[26] Graduate program - master track in software engineering. Accessed August 2016. https://www.csc.ncsu.edu/academics/graduate/degrees/se.php.

[27] Project email template. Accessed August 2016. https://github.com/SoftwareEngineeringToolDemos/Paper/blob/master/email.template.pdf.

[28] Task force on data, software, and reproducibility in publication. Accessed August 2016. https://www.acm.org/data-software-reproducibility.

[29] Virtual machine grading rubric. Accessed August 2016. https://github.com/SoftwareEngineeringToolDemos/Paper/blob/master/vm.rubric.xlsx.

[30] Virtual machine script grading rubric. Accessed August 2016. https://github.com/SoftwareEngineeringToolDemos/Paper/blob/master/vm.script.rubric.xlsx.

[31] Result and artifact review and badging. Accessed August 2016, June 2016. http://www.acm.org/publications/policies/artifact-review-badging.

IX. Appendix

Here are some random results that didn't fit elsewhere.

Several of the tools we analyzed were a service or had a service component; for instance, several tools ran as web applications on the creator's website. If the service is running, services provide a convenient way for users to try a tool. In total, for 8 tools, teams reported that the service was running when the teams tried to access it. For 2 tools, teams reported that the service was not running, but the researchers got the service running after the teams emailed them. For 4 tools, teams reported that the service never worked when they tried it.