Three Trillion Lines: Infrastructure for Mining GitHub in the Classroom

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

The increasing interest in collaborative software development on platforms like *GitHub* has led to the availability of large amounts of data about development activities. The *GHTorrent* project has recorded a significant proportion of *GitHub's* public event stream and hosts the currently largest public dataset of meta-data about open-source development. We describe our infrastructure that makes this data locally available to researchers and students, examples for research activities carried out on this infrastructure, and what we learned from building the system. We identify a need for domain-specific tools, especially databases, that can deal with large-scale code repositories and associated meta-data and outline open challenges to use them more effectively for research and machine learning settings.

CCS CONCEPTS

• **Computing methodologies** → *Machine learning*; • **Software and its engineering** → *Abstraction, modeling and modularity*.

KEYWORDS

Repository Mining, GitHub, TravisCI, Big Code, Teaching

ACM Reference Format:
Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2020. Three Trillion Lines: Infrastructure for Mining GitHub in the Classroom. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3397537.3397551

1 INTRODUCTION

The availability of large-scale datasets about open-source development activity, such as *GHTorrent*[3] and *TravisTorrent*[2], paves the way to study and train machine learning (ML) models on real data with the least sampling bias possible. Furthermore, it allows students to develop skills in large-scale data management and analysis while working on topics related to programming languages, software architecture, and development processes.

In collaboration with the students working on these datasets, we evolved a platform that allowed students and researchers to answer a wide range of research questions. Since this infrastructure is continually improved, we provide a short overview of the current state and limitations that might lead to future improvements. The questions we are addressing in this experience report are the following:

(1) What data are we using?
(2) Which server infrastructure can support around 10 concurrently working users?
(3) How can we automate large-scale data import?
(4) Which defects and failure modes need to be dealt with during import and operations?
(5) Which tools worked well for researchers using this data, and which types of research questions are being answered with them?

We conclude with a set of insights and research topics that were enabled through our infrastructure. We emphasize that, although the infrastructure is used for statistical analyses and training of a range of ML models, this instance is not yet equipped for GPU-based *deep learning* (DL).

2 DATASET

The dataset we focus on is a combination of several sources:

(1) A relational schema of meta-data provided by GHTorrent[3]
(2) Commit messages and file changes from GHTorrent's raw data[1].
(3) Actual repositories with complete directories and files cloned from GitHub
(4) TravisTorrent[2] built meta-data

GHTorrent Relational Schema. The relational schema contains linked meta-data about users, projects (repositories), commits, and other GitHub-specific activities like issues, pull requests, and comments associated with each artifact. Some content, such as comments, are present, while commit messages or file patches are not.

Individual File Changes. The *GHTorrent* project provides the collected raw data from the GitHub API in a document database (MongoDB). Due to resource constraints, we re-processed their database dumps to extract which file paths were changed in which commit, and collected the *patch* that encodes the full change to that file. This resulted in a new table with about 1.22×10^{10} entries that can be joined with the existing meta-data schema.

[1] http://ghtorrent-downloads.ewi.tudelft.nl/ (last accessed 2019-12-31)

Repositories. To complement the incremental view on the development process with actual file content, we use an infrastructure that downloads full repositories from GitHub, optionally checks out working copies for file-based tools, and indexes the file locations in database tables to allow SQL queries directly return file paths of interest. Content itself has to be retrieved by either reading a file in the working copy or accessing the repository using a Git library, e.g. `libgit2`.

A small subset of repositories has an associated tree of XML files that represent the *SrcML*[2] representation of the code. This allows XML-based tools to query and process their abstract syntax trees using *XPath* and *XSL transforms*.

TravisTorrent. The *TravisTorrent* dataset contains build reports from the continuous integration service *TravisCI*[3]. A single table containing all builds (incl. which commits where built and whether the build was successful) is part of our relational database.

Moreover, a subset of about 73000 builds of 20 active Java projects has been reprocessed down to test case level, i.e., for each built we know test failures, successes, and durations, and which commits have caused them. [8]

3 INFRASTRUCTURE

3.1 Hardware and Software Setup

Requirements. We designed our setup in a way that allows us to quickly add new capacity as the dataset grows, as we could only vaguely estimate required space upfront. The setup was to be operated on-premise and with free, open source software only as we intended to avoid vendor lock-in or high operating costs associated with cloud providers. We also intended to access the database at full network speed from the university network. We expect to serve up to 10 user sessions and require live fail-over capabilities and redundant disks.

Virtualized Infrastructure. Our lab runs multiple dual-socket servers and a disk array (12x 4TB SAS disks in RAID6 configuration). This physical infrastructure is shared with other research and teaching services, our dataset occupies a single VM. We use *XCP-ng* as hypervisor and all servers joined a single resource pool. VM storage consists of virtual disk files stored on the disk array. This so called *multipath* configuration allows live migration between physical servers as hardware downtimes due to failures or upgrades are unavoidable.

We use one VM with 64 CPU cores, 320 GB main memory, and 24 TB of virtual disks. It supports live-migration between physical servers.

Bandwidth between server and disk array can reach 1 GB/s, but averages at 400 MB/s. Although fast for disk-based storage, this is the limiting factor for querying the *GHTorrent* dataset.

The run-time of an SQL query can be estimated by adding sizes of scanned tables and divide by 400 MB/s. For our largest table, this is 8.5 hours.

[2]https://www.srcml.org/ (last accessed 2020-05-15)
[3]https://travis-ci.org/ (last accessed 2020-01-11)

VM Provisioning: CPUs and Memory. Most analysis jobs are not parallelized and the majority of analysis tasks is I/O-constrained, i.e., reads from disk, network, or runs database queries. We concluded that during initial student contact four CPU cores would suffice and later extended the VM to 16, then 64 CPU cores as we configured PostgreSQL to use additional workers per query. To allow students to keep long-running processes (incl. *Jupyter* notebooks) active over the course of a semester, we initially provisioned 128GB of main memory, later extending the limit to 320GB.

Our VM provisioning worked as expected. CPU power was almost never fully utilized, but disk I/O frequently reached bandwidth limits even with a single user session. Memory was usually not used more than 50 % by user sessions, but the database service indirectly benefited from spare memory as Linux keeps disk data in its *page cache*. When more than 100GB remain unused over a long period, querying the first few million rows of each database table becomes orders of magnitude faster. This behavior benefits explorative queries, e.g. where students sample a small subset of the data to test a query that eventually runs overnight.

Repository analysis is I/O-intensive. Generously over-provisioning main memory helps Linux cache disk data, but additional CPUs are rarely used.

Storage Provisioning. Storage requirements tend to grow over time, as GitHub data rapidly grows, and so does the *GHTorrent* dataset. Intermediate results should be stored in the database to allow re-use over expensive re-computation, e.g. as *materialized views*. Virtual disks can only grow to 2 TB, so we have 12 virtual disks allocated to our VM. They have to be added to a *volume group* using the Linux Logical Volume Manager (LVM). A single *logical volume* stretches over the full volume group and hosts an `ext4` file system.

A benefit of this setup is the ability to add a new virtual disk when needed, while LVM can allocate the new space to the mounted volume on-line. A major drawback was our choice of `ext4` over high-capacity file systems like ZFS, as we once approached the limit of *inodes*[9], i.e., the maximum number of file and directory entries.[4] We mitigated this problem by limiting the number of checked-out Git working copies, of which we temporarily hosted over 600 000.

Virtual disks in connection with logical volumes (LVM) allow VM disk space to grow over time. However, `ext4` limits the total number of files.

4 DATA PROCUREMENT AT SCALE

The *GHTorrent* project caches all GitHub API responses they ever retrieved, including every commit, using MongoDB as document storage. They export this data using MongoDB's native BSON (Binary JSON) format and provide compressed `.tar.gz` archives as downloads[5]. These dumps are incremental, the first four being created in larger intervals (the largest was 5.1 TB in a single `commits.bson` file), followed by 1252 daily increments[6].

[4]About one billion in our file system
[5]ghtorrent-downloads.ewi.tudelft.nl (last accessed 2019-12-04)
[6]Until 2019-06-30

Table 1: Number of rows per relation. Bold are primary entities. Starred (*) relations have been extracted from the MongoDB dumps and can constitute a superset of the original relations, (**) were computed from existing tables and do not constitute relations. Arrows (→) indicate that these relations link their entity to other entities.

Entity / Relation	Count	Entity / Relation	Count
Users	32 411 734	**Issues** → Projects, Users	98 076 172
Followers → Users	29 809 738	Iss. Comments → Users	148 429 082
Org. Members → Users	681 054	Iss. Labels	27 474 913
Projects → Users	125 486 232	Iss. Events	136 108 876
Proj. Commits → Commits	6 251 898 944	**Pull Requests** → Projects, Users	52 018 443
Watchers → Users	150 035 336	PR. Comments → Users	35 453 290
Proj. Members → Users	12 618 714	PR. Commits → Commits	266 030 349
Proj. Languages	138 205 530	PR. Events	135 081 995
Proj. Topics	517 318		
Commits → Users	1 368 235 072	**TravisCI Builds**	3 702 595
Commit Messages*	1 406 641 206		
Parents → Commits	1 365 342 872		
Parents*	1 437 712 365		
Commit Comments	5 682 741		
Patches*	12 212 981 402	Lines Added/Deleted**	3 158 855 923 812
Database size on disk	14.72 TiB		

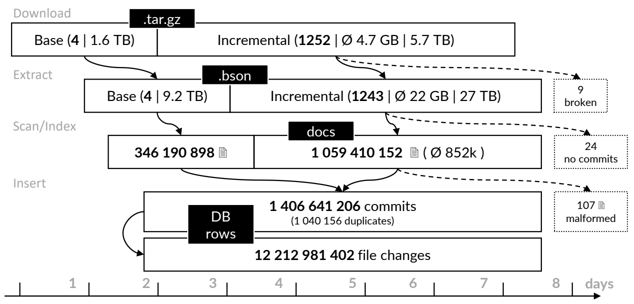

Figure 1: Visualization of the import pipeline with quantities and time scales. Timing is approximated and depends on network bandwidth.

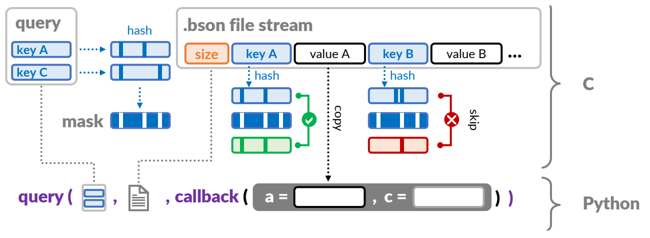

Figure 2: Working principle of the streaming BSON library using bloom filters and Python callbacks.

Our relational database of choice is *PostgreSQL* (Version 12.0 at the time of writing, 9.4 initially). We ruled out the option to directly import the data into MongoDB for two reasons:

- The majority of values are full URLs pointing to linked resources. We were primarily interested in data that is usable offline (such as author, message, and file changes of a commit).
- The reliability of `mongorestore` proved insufficient in multiple experiments to reproducibly import dumps from a foreign source.

Filtering desired information from files that large cannot be achieved using traditional BSON libraries, as they parse the full document when only a small subset of keys is needed. This prompted us to develop our own solution to *query* and import large BSON files efficiently.

BSON Query Engine. Our BSON query engine is a Python extension written in C. It consumes a (nested) list of *keys* to extract, a file descriptor, and a Python callback function with arguments that match the keys to be retrieved. The engine scans the consecutive documents in the BSON file stream and only parses values that correspond to a key in the query. Key lookup is accelerated by *bloom filters*. Querying is recursive, e.g., for a sub-document we can issue a sub-query for the value's elements of interest. Parsed values are handed back to the Python interpreter by passing them as arguments to the callback function.

In our main use case, a Python script sanitizes the individual values and insert them into a database, but we have used the library extensively to explore the large MongoDB dumps and gain insights into what keys and values to expect.

Batched Import. To better recover from crashes, parallelize imports, and allow users to interact with the database while import scripts are running, we import batches of 1024 documents per transaction. A failure in a batch causes a roll back and writes the offending document offsets to disk, so we can manually re-import this batch after resolving the issue.

Parallel Pipelines and Bus Bunching. To better utilize resources, we operate the import process as a pipeline processing one downloaded archive at a time and deleting successfully imported archives. In our current setup, four instances of the pipeline operate in parallel.

However, parallelizing the import suffers from an instance of the *bus bunching problem*[7], where multiple processes cycling through limited resources (e.g. network during download, disk during extraction) slow each other down when using the same resource. This causes more processes to "catch up" and get trapped in the same phase as the other processes, slowing them down even more. The first process to exit a resource-constrained phase (e.g. completing download and beginning to extract) will cause the other processes to exit faster, so they catch up again and clump in the next phase (e.g. exhausting disk I/O while extracting). This effect of processes syncing up at the same resource increases with more processes and can only be addressed by a different architecture in the future.

Bus Bunching: If a long-running task cycles through constrained resources (disk, network, CPU), starting many instances of this task causes them to eventually bottleneck themselves at the same resource, leaving the remaining resources idle.

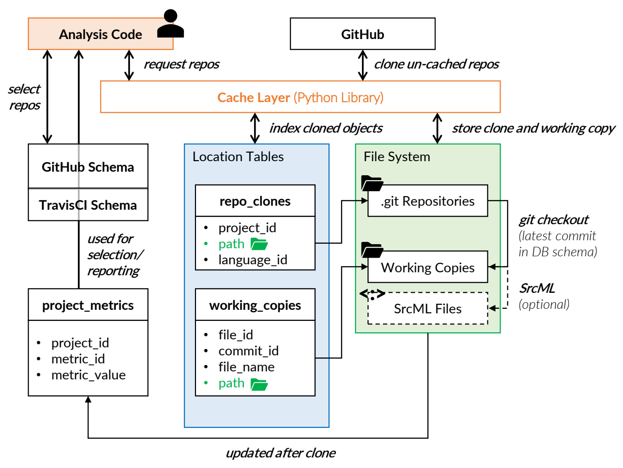

Figure 3: Repository management infrastructure linking file system and GitHub database

4.1 Failure Model

Our failure modes are dominated by encountering invalid data or system limitations (see Table 2). In the irrecoverable case, knowing which or how much data is missing helps assessing the validity of results based on that data. Automating the import to a degree that requires minimal intervention – including recovering from crashes – makes it easier to update the dataset or fully rebuild it. We describe problems in detail to help avoid them in similar setups:

Data Level. The original data contains dubious UTF-8 encoding (e.g. UTF-16 surrogates, humorously known as WTF-8 encoding, or multi-byte null characters), is incomplete (e.g. API responses that classified a file as `renamed` without stating the previous filename), or contains otherwise unexpected data (e.g. timestamps from the far future). We can only speculate about their cause, but must consider the possibility that commits or repositories have been created or manipulated using old [8] or alternative Git implementations with relaxed or even malicious treatment of timestamps and filenames. During import, we sanitize UTF-8, while timestamps are loaded as text to let analysis code decide how to parse them and which ranges be acceptable. The timestamp `0000-00-00 00:00:00` is invalid and regarded as null.

When dealing with CSV, numeric columns can contain booleans (e.g. `TRUE`), or be escaped (e.g. `"42"`). Missing values can occur as the empty CSV cell, `\N`, `NA` (specifically in TravisTorrent), or the empty string `""` in non-string columns. We replaced the common cases (such as the different variants of null) using the Linux tool `sed` before further considering the CSV file.

Expect source data (unicode, dates, numbers, booleans, and null) to be laxly encoded. Expect target languages and databases to be strict about types and their encoding.

Distribution Level. Hosted `.tar.gz` archives containing the exported `.bson` document collections can be corrupted or truncated, causing either the `tar` tool to fail or our importer to encounter invalid documents in the BSON stream. We lost 9 of the 1252 incremental archives to data corruption (i.e., 1 in 139), and the successfully extracted `commits.bson` files contained 107 defective documents (i.e., 1 in 13 million)[9]. Using the other archives, we estimate that about 7 to 8 million documents are currently missing from our dataset, which amounts to 0.5 % of all commits.

Network, OS, and Disk level. Faults during the download were transient and fully recoverable using fast timeout, unbounded retries, and the continue flags in `wget` [10]. Unexpected failures have occurred and prompted us to keep data import resumable and its progress recoverable through fine-grained logging. If started from a previously interrupted state, our import scripts can resume from the most recently logged record. Serious issues, such as a failing host bus adapter connecting the disk array, can still lead to inconsistencies between what the import tool recorded and what the database was able to recover, causing our script to accidentally re-insert about one million commits. A deduplication of the dataset is no longer possible, as the required sorting or hashing stage needs more space than physically available.

Transactions, progress logging, and the ability to restart from any logged progress leads to a *resumable* design that saves time after failures.

[7]Named after the effect that late buses picks up more waiting passengers, making them even later and causing spaced vehicles to catch up

[8]Git before `1.7` did not handle non-ASCII filenames uniformly across platforms.

Table 2: Critical defects and failure modes encountered during infrastructure setup.

Item	Offending cause	Observed effect	Prevalence	Current mitigation
Strings	UTF-16 surrogates in UTF-8	rejected by DB	≈ 1 in 500k strings	substitute by replacement character `U+FFFD`
	Multi-byte null	run-time error in Python	–	replace by single-byte `U+0000`
	Wrong type in CSV column	rejected by DB	–	replace common cases with `sed`, regard rest as null
Dates	zero date (0000-00-00 00:00:00)	rejected by DB	≈ 1 in 100M records	replace by null
	in far future or past	unexpected behavior during analysis	≈ 1 in 1B records	retain as text, ignore during analysis
BSON	invalid document	parsing error	≈ 1 in 13M records	skip individual document
Archive	truncation/corruption	`tar` terminates with unexpected EOF	9 of 1252 archives	skip archive
VM	disk space exhaustion	download/extract/import stops	≈ annually	attach new virtual disk file, expand volume group
	inode exhaustion	file creation no longer possible	–	limit checked-out Git working copies
Host	server failure	operation stops	–	reboot VM on other host
Disks	disk failure	availability loss, data loss	–	RAID 6 configuration
	RAID or HBA failure	kernel panic, data loss	–	Write-ahead logging (WAL)

5 TOOLS AND EXPERIENCE

Working on the Server. We use *Jupyter Lab*[11] as a tool to program and run Python on the server. Each user account is set up with a configuration that runs a password-protected Jupyter Lab on a personalized port via HTTPS. The UI allows them to navigate and edit all files (including notebooks) in their home directory, while all users can programatically access shared locations, such as the directory of repositories, and connect to the database. Using long-running *notebooks* is popular, as complex queries can be left running overnight and re-visited from another computer.

The capability to continue exploration over multiple days from any location has its benefits, but often results in code that is extremely difficult to re-run, e.g. after a crash or in a reproduction attempt. Moreover, we observe that almost no tests are being written for software artifacts ("notebook-driven development") and collaboration is difficult. Groups tend to diverge quickly, each performing exploration in their own notebook, since sharing of notebooks or results is hard.

> Server-side Jupyter notebooks greatly benefit data-heavy exploration tasks. However, the resulting collaboration and code quality needs improvement ideas.

Managing Repositories and Working Copies. To manage repositories alongside the relational database, students built a Python library for *cloning* the repository from its original location while storing its disk location in the database (see Figure 3). A commit can be checked out into a directory, and the files in this *working copy* will be indexed in the database as well. The library computes a number of metrics (e.g. lines of code) to allow future users to quickly retrieve or rank repositories by that metric.

This abstraction layer accumulated more than 600000 repositories, which led to performance degradation of the `ext4` filesystem; directory-based tools like `ls` or `du` became unusable and *inode* limits were nearing exhaustion. Currently, we encourage the use of *GitPython* to access *bare* repositories without checking out working copies.

Parsing Code. Up to now, most users resort to parsing code in the respective host language (e.g. using Python's `ast` module), or use *SrcML*, which can generate an XML view on the source files. Python's `lxml` is a popular choice to query the results with XPath or transform them via XSLT. Examples include generating a GraphML representation of a call graph to be visualized in Gephi[1] or computing code metrics using XPath queries (number of classes, methods, dependencies, arguments, etc.)

> XML-based tools like SrcML, XPath and XSLT are verbose, but flexible and fast tools for code analysis.

However, SrcML could not keep up with language evolution. In the future, we plan to evaluate Tree-Sitter[12] and Babelfish[13], both are frameworks for parsing a wide range of languages.

We perceive a lack of persistent structures, e.g. a binary alternative to SrcML, which helps to semantically query a program and all its versions at a larger scale and allows intermediate results to be stored at node-level for future re-use.

Learning Models. Tools used for analysis and machine learning are extremely specific to the use case at hand. In general, the ecosystem provided by Python's *NumPy* and *SciPy* is sufficient for most

[9]We did not inspect other `.bson` files in the incremental archives
[10]`wget -c --retry-connrefused --timeout=5 --tries=0`
[11]https://jupyterlab.readthedocs.io (retrieved 2019-01-14)
[12]https://tree-sitter.github.io (Retrieved 2019-01-14)
[13]https://docs.sourced.tech/babelfish (Retrieved 2019-01-14)

descriptive tasks, while *scikit-learn* is a popular choice for predictive models. Although supported, the *TensorFlow* ecosystem remained an unpopular choice, as its data and programming model does not align well with the tree-like and graph-like models occurring in repository analyses.

5.1 Research Questions

We provide a small selection of research questions and how they are currently approached using our platform.

Cross-language Effects. In our seminar, we address the question how experience in one programming language affects the quality of code written in a different language. As an example, Horschig et al. [4] measured correlation between code quality issues in Python and the original language of the programmers causing them (e.g. C++ programmers leaving semicolons at the end of lines in Python). We continue to study how picking up a new language affects coding style in one's original language. The platform supports these types of research, as students can select candidate contributors by a wide range of metrics and isolate their contributions in actual source code.

Maintainability. We try deriving insights from development history about the maintainability of the underlying system. For example, participants of our seminar explore how modularity can be measured based on co-edited locations, co-authorship, test results, pull requests, or issue discussions. We extensively used the database to gather candidate projects and evaluation datasets for regression test prioritization [7, 8].

Tooling. We explore tool designs at two levels:

(1) Tools to help users analyze the dataset itself, such as live programming environments for the dataset,
(2) Tools that use the dataset to help their users program better, such as recommenders and linters. We are primarily using Lively4 [6] and Squeak/Smalltalk [5] to build tools in an exploratory fashion.

Technical Problems. Apart from data-driven research questions, we are interested in exploring high-performance data structures and algorithms that support above research questions. These range from databases specialized to support syntax trees and version histories, over data structures for ASTs, to fast code metrics that scale well. We see our dataset as an opportunity to evaluate such designs, and at the same time would be their initial users. Git repositories are built for collaboration, not for analysis – relational databases are built for analyses, but not for code. Hence, we lack technical solutions in their overlap.

6 CONCLUSION AND FUTURE WORK

With our enriched GHTorrent dataset, theoretical limits of file systems, databases, or libraries unexpectedly turn into practical barriers. Moreover, "one-in-a-billion" faults turn into frequent events if not dealt with by robust and resumable design.

Instead of scaling out, we developed custom tools and ways of working that allowed us to process "Big Code" on a single VM. This setup was not an upfront design decision but emerged as we observed how students dealt with these challenges.

Working with the setup, we experienced a mismatch between how databases and "Big Data" operate and how source code is organized and accessed through tools. While the former does not match the structure of software repositories, the latter does not scale to the same degree. A *database for code* could be of help here. Finally, with the availability of deep learning hardware, we look forward to address a completely new set of infrastructural and technical challenges in the foreseeable future.

ACKNOWLEDGMENTS

This research has been supported by the Federal Ministry of Education and Research of Germany (BMBF) in the KI-LAB-ITSE framework (project number 01IS19066).

REFERENCES

[1] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. 2009. Gephi: An Open Source Software for Exploring and Manipulating Networks. http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories.*

[3] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[4] Siegfried Horschig, Toni Mattis, and Robert Hirschfeld. 2018. Do Java Programmers Write Better Python? Studying off-Language Code Quality on GitHub. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming - Programming'18 Companion.* ACM Press, Nice, France, 127–134. https://doi.org/10.1145/3191697.3214341

[5] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97).* ACM, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[6] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a live development experience for web-components. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience, PX/17.2, Vancouver, BC, Canada, October 23-27, 2017.* 28–35. https://dl.acm.org/citation.cfm?id=3167109

[7] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *Programming Journal* 4, 3 (2020), 12. https://doi.org/10.22152/programming-journal.org/2020/4/12

[8] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *Proceedings of the Conference on Mining Software Repositories (MSR) 2020.* To Appear. https://doi.org/10.1145/3379597.3387458

[9] Andrew S. Tanenbaum. 2007. *Modern Operating Systems* (3rd ed.). Prentice Hall Press, USA.