Software effort estimates vs popular developer productivity metrics: case study of empirical correlation

WILLIAM BATES HARDING
Published February 2021
Last updated March 3, 2021

Abstract

A growing number of “developer productivity” tools[1] seek to use git metrics like “Commit Count” and “Lines of Code Changed” to help managers understand the work happening on their development teams. Graphs of “Commit Count” and “Lines of Code” have become ubiquitous across popular git hosts like GitHub, GitLab, and Bitbucket. It’s tempting for Managers to see the graphs these providers offer and extrapolate judgement about the performance of their developers, but little is known about the extent to which either metric correlates with “difficult work completed,” or if they correlate at all[2].

In this study, we use a large empirical dataset (n=2729) to report the correlation between a software team’s effort estimation (i.e., “How difficult will it be to complete X task?”), compared to three git-based software metrics: Commit Count, Lines of Code Changed, and Diff Delta. We find that all three metrics have some level of Pearson correlation with the development team’s difficulty estimation. The most correlative metric was Diff Delta, ranging from 26-63% in large repos (n=100+ tasks) analyzed. The corresponding r^2 observed for Diff Delta ranged from 7-38% in large repos.

A software metric that exhibits high correlation with effort estimation could facilitate novel opportunities across software companies. For Product Managers, such a metric could reduce busywork calculating “sprint velocity.” For CTOs, it could identify where tech debt accrues to a repo, to intervene before it debilitates their project. For CEOs, it could inform long-term plans by predicting how much work a particular team of developers will complete per time unit.

---

[1]Summary of five separate software metric analysis tools that have launched between 2016 and 2021: https://www.gitclear.com/pluralsight_gitclear_pinpoint_code_climate_code_development_kpi_metric_alternatives

[2] “Although the choice of these approximations is critical for the performance of the prediction models, there is no empirical evidence on whether LOC is actually a good approximation.“ --Shihab et. al, Is Lines of Code a Good Measure of Effort in Effort Aware Models

Analysis Method

The goal of this study is to evaluate the correlation between a team's collective judgement of how much effort will be needed to complete a task, compared to three available software metrics derived from a team's git commit history.

Story Points represent effort and difficulty

To represent a team's collective judgement of effort estimation, this study uses Story Points, extracted from Jira after being estimated by the team. As explained by Atlassian[3] in the top-ranked Google result for "Story Point Estimation,"

> "Story points are units of measure for expressing an estimate of the overall effort required to fully implement a product backlog item or any other piece of work. Teams assign story points relative to work complexity, the amount of work, and risk or uncertainty."

Another of the top-three Google results for "Story Point Estimation," from VisualParadigm.com[4], echoes Atlassian's description:

> "A story point is a metric used in agile project management and development to estimate the difficulty of implementing a given user story, which is an abstract measure of effort required to implement it. In simple terms, a story point is a number that tells the team about the difficulty level of the story."

These and other sources concur that teams assign story points by collectively judging the overall effort required to complete a task.

Story Points are relative to the team

There is no single scale on which Story Points accrue. This is by design: as described by Atlassian's Story Points article[5], "each team will estimate work on a slightly different scale, which means their velocity (measured in points) will naturally be different. This, in turn, makes it impossible to play politics using velocity as a weapon."

This means that, while every team will assign a higher number of Story Points when a task is perceived as "more difficult," there isn't expected to be a single constant *c* that could translate

[3] https://www.atlassian.com/agile/project-management/estimation
[4] https://www.visual-paradigm.com/scrum/what-is-story-point-in-agile/
[5] https://www.atlassian.com/agile/project-management/estimation

what “5 Story Points” represents for Team A vs Team B. Every team will have its own value of *c*, based on who is choosing the Story Points. Thus, the correlation between Story Points and code metrics must be evaluated on a per-project basis, where a consistent evaluation of Story Points can occur.

By evaluating correlation on a per-project basis, the noise of differing *c* values is eliminated from the final correlation analysis. To calculate the cumulative level of Pearson and r^2 correlation, this study uses a weighted average that combines the correlation value per project with the number of data points supporting that correlation (i.e., the number of issues analyzed in the repo).

Data collection criteria

The dataset provided below is sourced from more than 2,500 issues across 61 different git repos analyzed by GitClear[6]. The repos included were selected based exclusively on three criteria:

1. Signed up for GitClear and connected their repo to a Jira instance
2. Opted in to sharing anonymized data in their GitClear settings
3. Had specified, or could heuristically have deduced, which column in the Jira integration corresponded to Story Points

No further criteria were applied in selecting repos. The full set of data provided covers all 61 repos that meet these three criteria as of February 2021. In the Appendix, one can find the source code that was used to generate the CSV file, which was then imported into the Google Sheet. This source code shows that the data set provided is the full sample of repos in the GitClear database that match the conditions above.

Since no filtering is applied to pre-select these repos, there is a high variance in the correlation levels on a per-repo basis. This variance reflects expected differences in how random, real world teams assign Story Points to their tasks. The high variance observed should contribute to these findings being reproducible for future experiments, even with large, somewhat messy, real-world datasets.

Source code used for analysis

The full source code used to generate the CSV data set can be found in the Appendix.

[6] Source code analysis company providing this dataset free of charge for public use https://www.gitclear.com

Purpose

A software team with high correlation between their effort estimation and a software metric could enjoy decided advantages over competitors lacking such data.

Reduce time/busywork calculating “sprint velocity”

According to LucidChart’s popular article on Story Points[7],

> “You may at this point be wondering how many story points a team can complete during a sprint. That amount is called sprint velocity, and unfortunately, there’s no way to determine that until the first sprint has been completed.”

To the extent that a developer metric correlates with Story Points, the total Story Points completed per sprint could be predicted by the measurement of that metric. Before a manager chooses to move a developer to a new team, they could estimate how much of a difference that move ought to make in the rate of backlog reduction for the team.

That is, if we manage a Developer named Bertrand who averages 5 commits per day, and we find that 10 commits correlates strongly with completion of 1 Story Point, then we can estimate that Bertrand should add 5 Story Points per 2-week sprint to the current output of the team. Of course, the *true* cost of adding Bertrand is also influenced by other factors, such as project assimilation costs for the new developer (i.e., understanding conventions and structure of the new repo), and increased communication costs for the existing team.

Augment technical interviews

The current recruitment process for software developers bleeds efficiency at every turn. Valuable Senior Developers, often including the CTO and VP of Engineering, are compelled to interrupt their project work to participate in long, arduous technical interviews. According to

[7] #2 Google result for “Story Point Estimation” as of February 2021
www.lucidchart.com/blog/how-to-estimate-agile-story-point

Google research, the best interview techniques yield somewhere between zero[8] and 29%[9] predictive ability in identifying the most talented candidates. Recruiters have to be paid 15-25% commissions per candidate referred. If a metric could be calculated locally on a developer's machine, analyzing the git repos they've previously contributed to, a team could hypothetically approximate how many Story Points this prospective developer would complete per sprint. This could be accomplished alongside a reduction to time spent by *interviewer* and *interviewee* on unpopular tasks like whiteboarded code questions, take-home coding projects, and the like. Which, again, still only yields 29% predictive confidence in the best case scenario[10].

In a more efficient system, the benefits of being a productive developer could be readily measured, and would ensure that developers with atypically high velocity were compensated accordingly.

Proactive help for struggling team members

In order to know when developers are struggling, it's common practice for Lead Developers to "check in" with their team a few times per week. The periodic check-in is an inefficient way to discover when a team member is struggling, given the "false positive" rate (the number of developers checked in vs number who needed help) tends to be extremely high. Most developers don't need help most of the time, and asking them if they need help breaks up their flow state, reducing progress[11].

It would be more efficient if a Manager could proactively detect that a teammate was struggling without needing to ask. A highly correlative git metric could let a Manager detect when their developers are floundering as soon as its output starts to drop.

Inverting this idea, Managers could use a highly correlative metric to evaluate when their Story Point estimates may have been poor. If their repo has higher variability in its Story Point correlation than industry averages, the Manager could evaluate whether the issue estimation process should be revised.

---

[8] "Years ago, we did a study to determine whether anyone at Google is particularly good at hiring. We looked at tens of thousands of interviews, and everyone who had done the interviews and what they scored the candidate, and how that person ultimately performed in their job. We found zero relationship. It's a complete random mess" --Laszlo Bock, Laszlo Bock, senior vice president of people operations at Google, New York Times, 2013

[9] "In 1998, Frank Schmidt and John Hunter published a meta-analysis of 85 years of research on how well assessments predict performance... The best predictor of how someone will perform in a job is a work sample test ([with r2 value of] 29 percent)." --Hire Like Google, Wired, 2015

[10] Hire Like Google, Wired, 2015 original research by Schmidt and Hunter at link

[11] https://stackoverflow.blog/2018/09/10/developer-flow-state-and-its-impact-on-productivity/

Detect how and where tech debt is created

If it's possible to map task complexity to a software metric, then it also becomes possible to identify what preceded the creation of complex code. That is, if a task estimated at a high value, like 11 Story Points, ends up taking 10 days to modify 20 lines in Directory A, then we could say that making changes in Directory A seemed to be very complex. Then we could look at the age and quality of the code that comprises Directory A, and make informed decisions about whether to take lessons that could help make future code more easily adapted.

Knowing which directories avail the lowest rate of Story Point completion helps a VP of Engineering or CTO calculate when large-scale refactoring is justified. Particularly if Senior Management can see that many upcoming features or bugs pertain to code located in a low velocity directory, the team can accelerate Story Point completion by seeking to remediate what makes the code in the directory slow/complex to adapt.

Long-term executive planning

Over a longer time period (e.g., a quarter, a year), a software metric with high Story Point correlation could project how much is likely to get done by a team of developers. After the effort for prospective tasks has been estimated, the Product Manager can use the conversion rate *c* to translate Story Points to their software metric of choice. Then, a team can be constructed that has a historical record of accumulating the software metric at a rate that should complete X number of Story Points by Y date.

For example, if 100 commits reliably correlated with 10 Story Points completed, and a team needs to maintain a cadence of 20 Story Points per week to reach its milestone, a Project Manager could assemble a team of developers such that its members cumulatively average 200 commits/week. The Manager could then feel confident the team will hit their weekly Story Point target, if the correlation between Story Points and Commit Count is high enough.

This informs how aggressive a CEO or Product Manager can be with their product roadmap, while keeping touch with a ground truth measurement. A metric with high Story Point correlation means the manager has instant feedback on whether future milestones are likely to remain on track as the developers on the team swap in and out.

Limitations of Method

In this section, we review known limitations of this study's design.

Variability in effort estimation efforts

For high Story Point correlation to exist, a team's leaders need to make consistently accurate estimates of how much effort will be required to complete a task. The stakeholders choosing the Sprint Points have to be "tuned in" to their project's existing tech debt, and they need to avoid letting politics influence the estimation.

If a team has limited information to make their effort assessment, their estimates will be more random. This randomness sets an upper limit on how closely Story Points might correlate with any software metric. The more the data set grows, the more randomness seeps in to drag down correlation values.

There are limited tools available to assess the quality of the team's estimation abilities, but the source code in Exhibit A of the Appendix illustrates one countermeasure this study used to control for atypical estimation: if **all three** of the developer metrics were negatively correlated with a repo's Story Points estimate, then we exclude that Repo's data points from the data set. As of February 2021, this requirement excluded less than 5% of all eligible repos.

"Lines of Code" correlation artificially boosted

The values of all three metrics are extracted from the GitClear database after having been processed by the company's "Commit Crunching engine" as described in the help documentation[12] for Diff Delta. In contrast to git stats generated from GitHub and conventional git hosts, GitClear identifies files and patterns that do not reflect developer work being done, and ignores the lines changed in these files. For example, every Ruby on Rails project contains a "structure.sql" file that is auto-generated by the Rails library any time the database is migrated. GitClear excludes this file from analysis, and so the lines of code that changed in that file are not included in the "Lines of Code Changed" count, even though a more basic interpretation (such as that afforded by all free git stat tools) would include these if "Lines of Code Changed" per issue were examined. Read more about how Lines of Code Changed is artificially increased in the Technical Details section of the Appendix.

Mapping commits to Story Points is non-trivial

The dataset may have some errors in how software metrics are associated with issues, and in turn, how issues are associated with Story Points.

To connect commits with issues, GitClear looks at the git commit record, seeking a reference to the task's external identifier. All issue data in this dataset comes from Atlassian's Jira, which allows a Story Points field to be added to the inputs when creating a new issue (e.g., Task, Story, To-Do item, Bug). To deduce how commits connect to tickets, data sources are analyzed

[12] https://www.gitclear.com/line_impact_factors

first by the content of their commit message, then by the content of their pull request title, and finally, by the name of the branch they were developed in. If any of these data sources yields a reference to a Jira ticket identifier, then the commit's metrics are associated with the Jira ticket.

In rare cases, commits will reference multiple Jira tickets, or will reference a Jira ticket that doesn't correspond with what is referenced in the pull request title. In these cases, we assign the commit's work only to a single issue, as described by the method above.

High tech debt, low correlation

Sometimes a task is complex not because it modifies large amounts of code, but rather because it is a small amount of unapproachable code that must be changed. Code that takes a disproportionately long time to modify is said to be unadaptable, or "harboring tech debt."

When code is well-documented, covered by tests, and separated into modules by concern, then energy required to adapt that code tends to remain constant as time passes. However, in real world projects, code is often authored under patchwork circumstances. This contributes to disjointed code lacking comprehensive tests and documentation. When this happens, the code becomes "complex," and the team will correctly estimate that changing a small amount of such code will take many Story Points, even though the measurable work to resolve the issue might only span a few changed lines.

For repos with large patches of tech debt, the correlation between "effort spent" and measurable developer output will be lower.

Results

Of the three metrics, over `n=2,791 repo issues` and `r=61 repos`, Diff Delta correlated most strongly with Story Points, establishing a Pearson correlation value of 38%. Commit Count correlated at 27% and Lines of Code Changed at 25%.

Diff Delta also demonstrated the highest r^2 correlation coefficient, with a 19% correlation coefficient, compared to 11% for Commit Count and 9% for Lines of Code Changed.

Result data summary

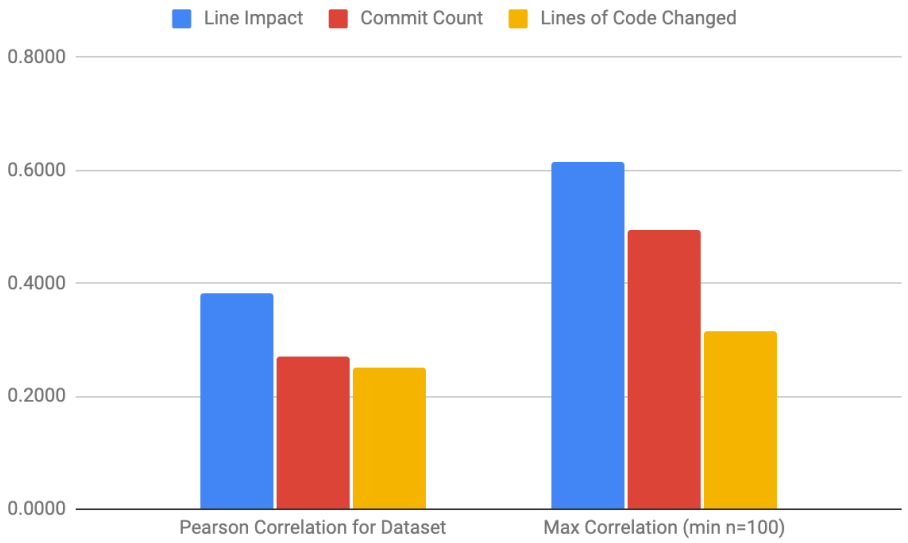
Find the full set of 2,791 issues analyzed in this spreadsheet and the Per-Repo Correlation Data in the Appendix. Here is the summary of across all repos analyzed:

	Weighted average		Large repos (100+ issues)			
	Pearson correlation	**r2 coefficient**	**Pearson Min**	**Pearson Max**	**r2 min**	**r2 max**
Lines of Code Changed	25.0%	8.5%	13.2%	31.4%	1.7%	9.9%
Commit Count	27.0%	11.5%	17.3%	49.4%	3.0%	24.4%
Diff Delta	38.3%	18.8%	25.9%	61.4%	6.7%	37.7%

The full analysis includes 61 distinct repos analyzed, and 2,729 data points. There were 5 repos that qualified for the “Large Repo” data group, these repos included 1,847 data points between them. As a reminder a “data point” is any issue that had Story Points assigned to it before being marked as resolved, per the “Data Collection Criteria” section in “Analysis Method.”

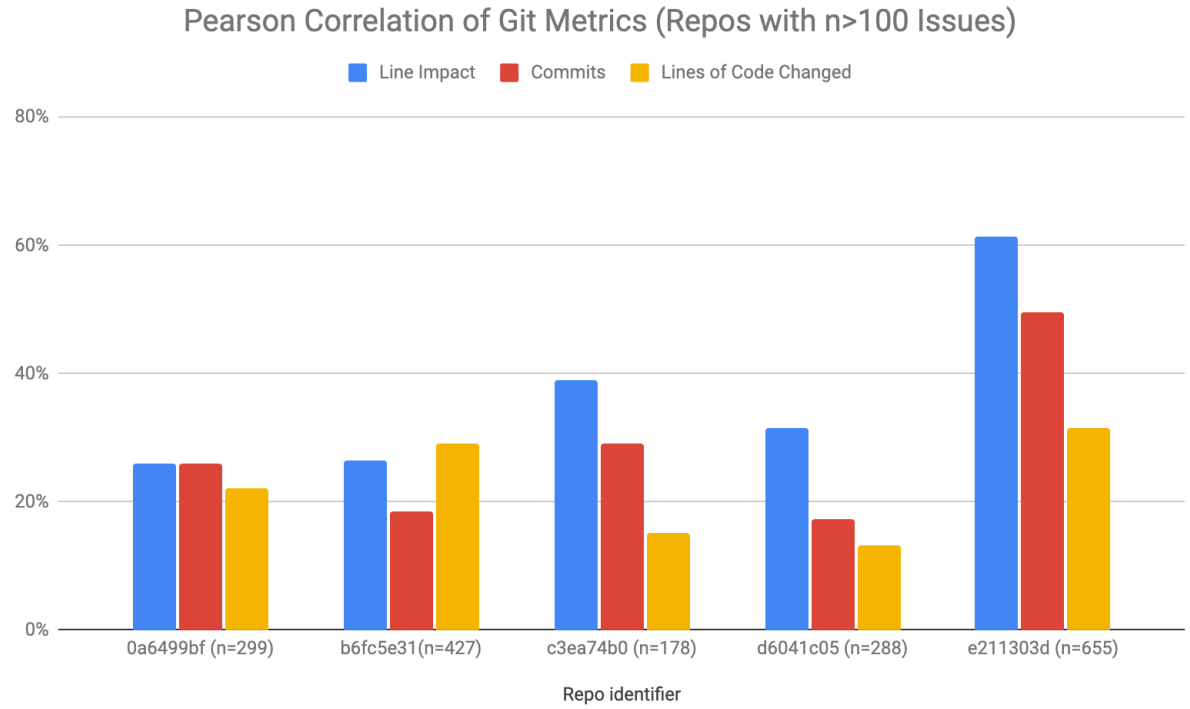
For both correlation calculations, the maximum correlation for a repo with n=100 issues is included as an indication for where the upper bound of correlation level might reside, given minimally random Story Point designation.

Here are the Pearson correlations visualized:

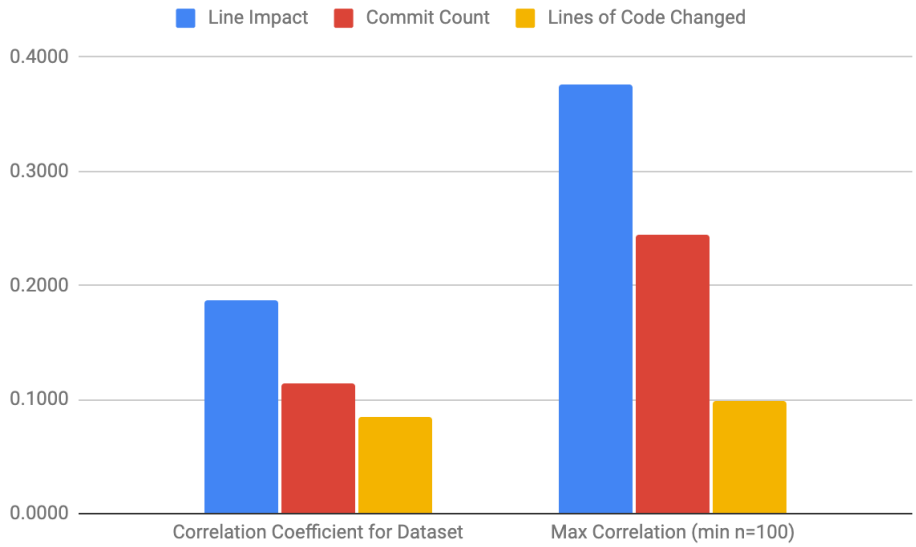


Average software metric Pearson correlation with Story Points, and max correlation among repos with at least 100 issues assigned Story Points

The Pearson correlation for the repos with n > 100

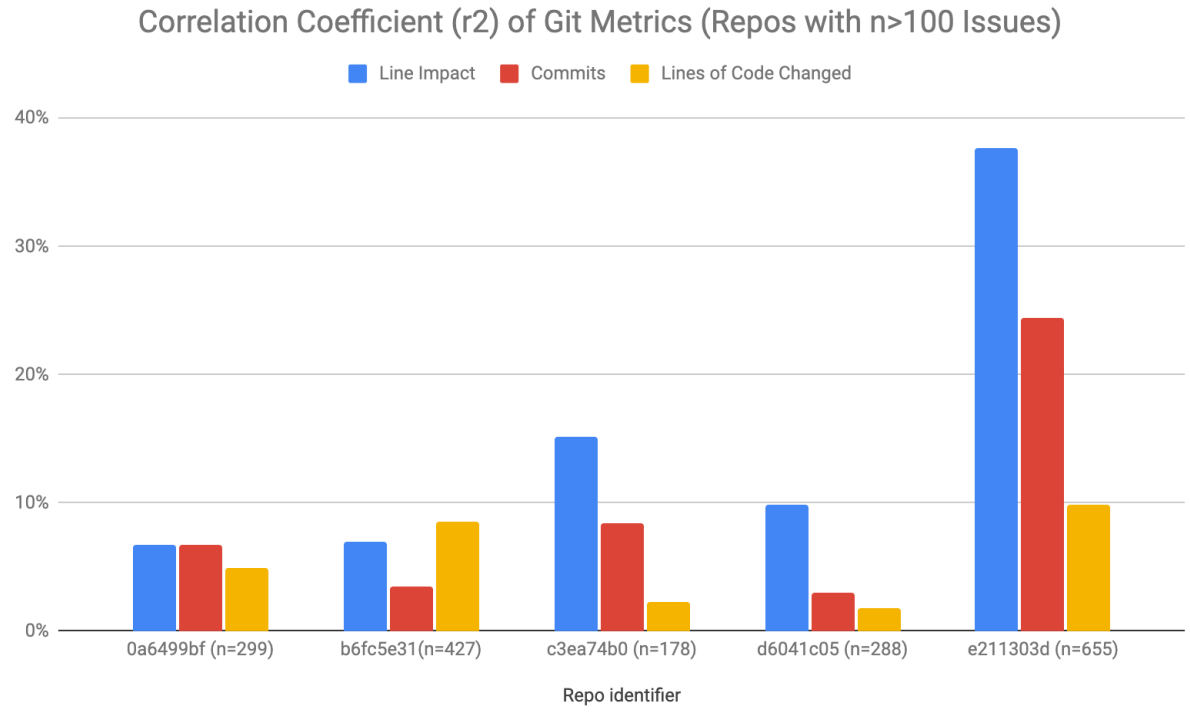


And the graphed r^2 correlation coefficients:



Average r^2 correlation coefficient for dataset, and correlation coefficient in maximally correlated repo with min n=100 issues having Story Points

Along with correlation in the repos with n > 100:



Interpretation

These data indicate that Diff Delta more closely approximates a team's estimation of effort than Commit Count or Lines of Code Changed. The improvement of Diff Delta over these conventional metrics ranges from +42% to +119%, with an average improvement of 70%[13].

The finding that Lines of Code have the lowest correlation with effort assessment mirrors previous results reported by Shihab, Kamei, Adams, Hassan[14], that "using LOC under-estimates the amount of effort required compared to our best effort predictor by approximately 66%." In spite of the metric's shortcomings, in the highest correlation repo, e211303d, Lines of Code still managed a correlation coefficient of 9.9%. This can presumably be attributed in part to the "Lines of Code" Correlation Artificially Boosted" Limitation previously discussed.

In aggregate, Lines of Code Changed produced a fairly low 8.5% r^2 value. This corresponds with conventional wisdom that Lines of Code is a dangerous metric to base decisions upon[15][16].

Commit Count performed second best in predicting complexity assessment, with an r^2 of 11.5% in aggregate, and 24.4% in the e211303d repo, where the Story Point estimation methods most closely paralleled observable code stats. Unfortunately, the utility of this correlation hinges on the Developer writing commits under the belief that their commit volume is irrelevant. Once they're cognizant that they are being measured, the metric loses all predictive value since Commit Count is the most trivial metric for developers to game[17].

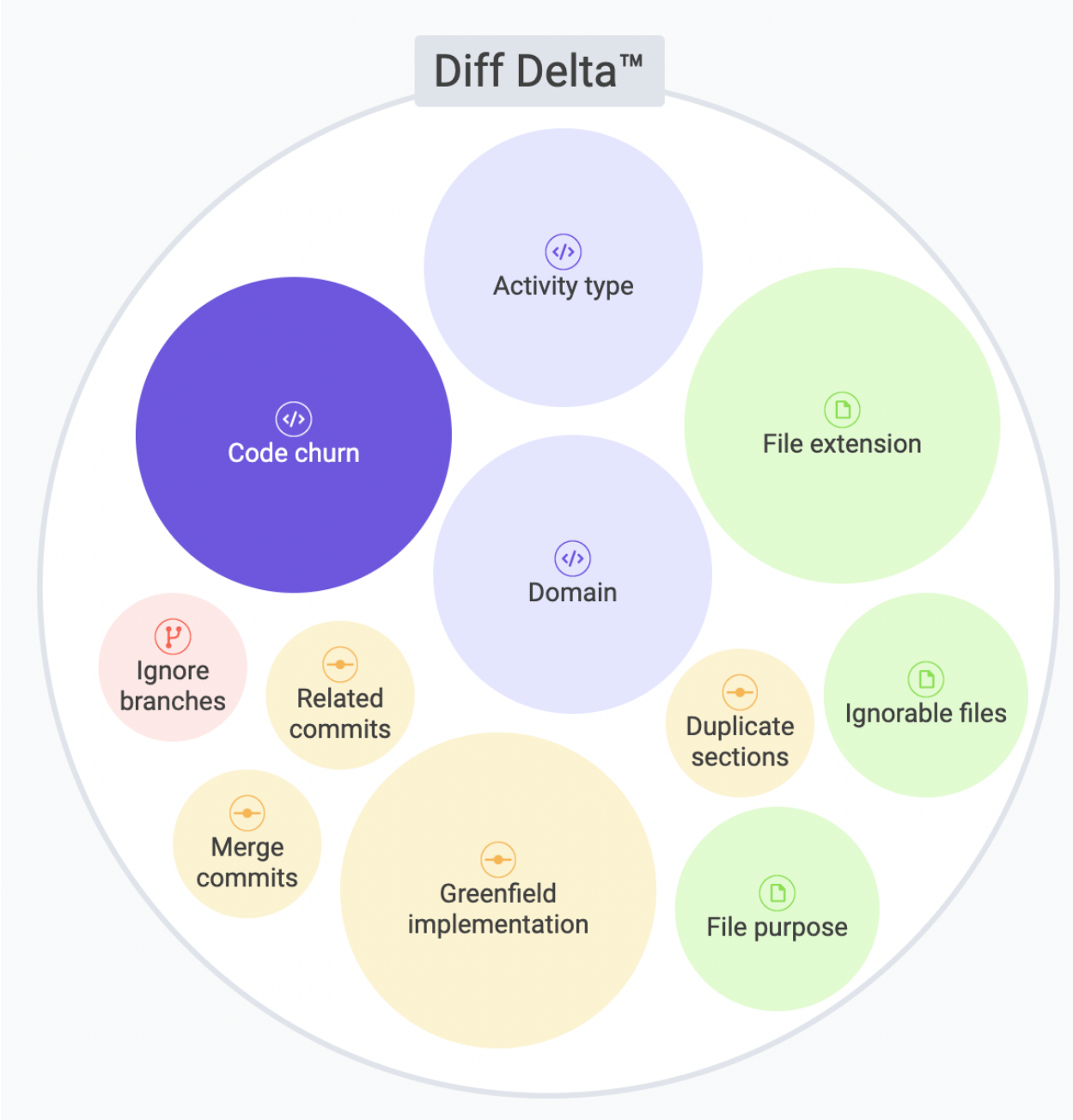
Diff Delta compiled a 38% Pearson correlation across the dataset, and a 18.8% r^2 weighted average. The largest repo in the dataset (n=655) suggests that a 38% r^2 Is possible at scale. The range of 19-38% suggests that evaluating a developer's Diff Delta would yield comparable predictive power as the most effective interview method available, the take-home project (r^2=29%).

Mechanism of Diff Delta correlation

Since Diff Delta correlates with effort more strongly than other metrics, it is worthwhile to examine the mechanics used to calculate Diff Delta.

GitClear provides the following diagram on its Diff Delta explanation page[18]:

[13] Diff Delta vs Commit Count Pearson/r2: 42%, 63%; Diff Delta vs LoC Pearson/r2: 53%; 120%. Average improvement: 69.5%
[14] https://www.sciencedirect.com/science/article/abs/pii/S0950584913001316
[15] https://www.gitclear.com/blog/the_4_worst_software_metrics_agitating_developers_in_2019
[16] "We asked a bunch of people why they hate the idea of measuring lines of code so much. The results here varied quite a bit, with people saying 'because people will just write lots of terrible code' or 'that's a very easy metric to game.' All of these are true of course."
https://www.pluralsight.com/blog/teams/lines-of-code-is-a-worthless-metric--except-when-it-isn-t-
[17] https://www.gitclear.com/popular_software_engineering_metrics_and_how_they_are_gamed
[18] https://www.gitclear.com/line_impact_factors



Approximate weighting of factors that combine to calculate Diff Delta

The company does not publish the source code for its evaluation. In the GitClear article[19] "Popular software engineering metrics, and how they're gamed," the company describes Diff Delta as a metric designed to measure cognitive energy:

> "Diff Delta is a metric designed to measure how much cognitive energy is being put into software development. This video offers an illustrated explanation of how that's possible.

[19] https://www.gitclear.com/popular_software_engineering_metrics_and_how_they_are_gamed

> The short explanation is that Diff Delta cancels out all of the interstitial activity ("churn") that happens as a feature gets developed, leaving a concentrated embodiment of the work that took place. Diff Delta is conserved across languages, so it takes a consistent amount of time to generate Diff Delta whether the developer is writing Java, Python, Javascript, or any other major programming language (30+ supported)."
>
> "In terms of business value, Diff Delta parallels Story Points: it illustrates how the cost of a task fluctuates depending on the developer to whom it is assigned. From this, an engaged manager can use the Domain Experts report to match Jiras to subject matter experts, which can dramatically accelerate product velocity."

Since Diff Delta was designed to assess cognitive energy expended, its target approximates Story Point effort estimation.

Developer metrics aren't enough

A team must not rely solely on any particular developer metric as the "single source of truth" in evaluating developer performance. Rather, developer performance is best thought of as the combination of many factors: *quantified metrics*, *collaborative spirit*, *domain expertise*, *mentorship ability*, and other factors. Just as no qualified manager would evaluate a developer solely based on their Story Points completed in a year, neither would a manager be wise to believe that any single metric will offer a full picture of the value created by a particular contributor.

Follow-up Research

We hope that future researchers will review the GitClear dataset and suggest areas that would be interesting to explore in follow-up studies. The more sources can reproduce our correlation results, the more confident we can be that they apply across different development contexts (e.g., startups, mid-sized companies, enterprise).

The biggest opportunity would be to increase the number of data points to more than 10,000. We hope to produce such a study sometime in 2021 or 2022, and would be eager to collaborate with researchers who have taken on work of this scale in the past.

An area that we remain especially interested to explore is the differences in individual consistency from week-to-week, across each software metric and Story Points completed. It seems reasonable to hypothesize that the number of Story Points completed from week-to-week would have less variability than any software metric, since Story Points are assumed to have the highest correlation with the amount of work to be done. Metrics whose

weekly variability is lowest would hold promise to further enhance the accuracy with which executives and Project Managers can approximate their project release schedule.

Finally, we would like to see follow-up research better separate the full “Lines of Code,” as interpreted by raw git, from the advantaged “Lines of Code” known to GitClear. That is, if the study simply used raw lines of code changed, as would be observed viewing GitHub Stats, it’s presumed that the Line of Code correlation would fall appreciably. Since 30% of all Lines of Code Changed are simply moved lines[20], these would inflate the metric but have zero correlation with complexity estimates.

[20] Stats on the extent to which Lines of Code falls into different categories of noise: https://www.gitclear.com/lines_of_code_stats

Appendix

Technical Details

Limitations of Method: Lines of Code Follow-up

The correlation of the “Lines of Code Changed” metric in this study is expected to be inflated over what might reproduce in subsequent analyses. A few file types whose lines of code are automatically ignored by GitClear:

- Auto-generated files (e.g., yarn.lock, gemfile.lock, structure.sql)
- Minified Javascript files
- Any files whose extension does not correspond to a known programming language
- Other directories manually marked by customers as non-relevant (e.g., vendor and library directories)

Furthermore, there are a few types of operations that GitClear recognizes and translates into what is measured as a single “Line of Code Changed”:

- Find/replaced code
- Updated code
- Moved code

These three operations comprise more than 30% of all lines that change, so when git providers treat these operations as two changed lines (as all but GitClear do), the correlation of “Lines of Code Changed” and task complexity is likely reduced.

Per-Repo Correlation Data

Below you can find the 61 repos that were analyzed for Story Point correlation.

Pearson correlation

Here is the full set of Pearson correlation observed per-repo, with repos of n=100 or greater **set apart in bold** since these data points are the most robust, and contribute most strongly to the weighted average of the dataset:

Repo pearson correlations	Record count	Diff Delta	Commits	Lines of Code Changed
04bffefd Pearson	6	-0.31937	0.17170	-0.40495
07a6a2fa Pearson	3	-0.21838	0.62361	-0.46038
097b4331 Pearson	12	0.52366	-0.12524	0.17671

0a6499bf Pearson	**299**	**0.25889**	**0.25936**	**0.22203**
0c492158 Pearson	26	0.22353	0.08266	0.05783
0f2aaf30 Pearson	12	0.23511	0.03633	0.16989
1353a4c4 Pearson	24	-0.02670	-0.12653	0.13393
17730c68 Pearson	8	0.07815	0.03939	0.14068
1d88ad13 Pearson	40	0.20626	-0.01764	-0.01458
2b682982 Pearson	11	0.80441	0.43239	0.80612
2cfcbf23 Pearson	27	0.37282	0.34018	0.33808
2fadc9df Pearson	16	0.36792	0.11054	0.29969
30a64ca0 Pearson	10	0.67743	0.38716	0.30210
3242485d Pearson	8	0.30877	0.35461	0.37715
3a218db3 Pearson	46	0.67413	0.47113	0.44695
3c8d9e25 Pearson	11	0.69774	0.84989	0.88054
3ed5dcff Pearson	44	-0.16362	0.04120	0.05688
472bc4dd Pearson	13	0.50252	0.72884	0.67365
4acd4acf Pearson	3	0.16559	0.15430	0.45343
4d69f0ac Pearson	22	0.24478	0.32123	0.19333
50315b0e Pearson	5	0.98115	0.99291	0.99940
50a0767b Pearson	48	0.16258	0.39594	0.11336
5a9ebe3c Pearson	4	0.87088	0.48507	0.93505
5ffac53b Pearson	19	0.19211	0.03199	-0.20661
606dfa8b Pearson	9	0.28914	-0.19414	0.07278
66779d5f Pearson	5	0.77292	0.68063	0.76713
674f7b2d Pearson	9	0.29326	0.38903	0.44908
72a585e3 Pearson	8	0.55238	0.28734	0.34115
752b1dd0 Pearson	30	0.22834	0.06030	0.21315
75305ee2 Pearson	10	0.26098	0.28385	0.27004
7e1f5915 Pearson	54	0.29643	-0.00168	0.18859
7e9398b4 Pearson	8	0.14901	0.11438	-0.04556
7ef3d268 Pearson	12	0.72787	0.40653	0.58152
80abc79a Pearson	3	0.45145	0.79259	0.49903
80c6eedd Pearson	4	-0.60806	-0.60612	0.81780
94aefe51 Pearson	15	0.34098	-0.06126	0.18911
9a7f05e0 Pearson	4	0.58738	-0.28521	0.23430
9b4ceb2f Pearson	3	0.75965	0.53444	0.63610

a90b4fc9 Pearson	6	0.24625	-0.17252	0.21033
ab5f83c9 Pearson	4	0.38738	-0.22268	0.08689
acf55047 Pearson	20	0.59130	0.68691	0.51041
b5078827 Pearson	12	0.32632	0.36575	0.21961
b6d9794a Pearson	13	0.14214	0.20464	-0.01495
b6fc5e31 Pearson	**427**	**0.26496**	**0.18533**	**0.29149**
b7260273 Pearson	9	0.10923	-0.04017	-0.16144
bcc4da08 Pearson	5	0.27923	-0.33436	0.39991
c08ca054 Pearson	10	0.38398	-0.49646	0.14637
c3ea74b0 Pearson	**178**	**0.38893**	**0.28997**	**0.15142**
c4abd096 Pearson	12	0.49558	-0.08717	0.63537
c556a08a Pearson	48	0.11769	-0.17979	0.09130
caa66f10 Pearson	26	0.44615	0.13578	0.34608
cf472176 Pearson	17	0.47926	0.30832	0.40760
d231b61d Pearson	7	0.43732	0.49014	0.50620
d4005b42 Pearson	3	0.99911	0.99926	0.99946
d6041c05 Pearson	**288**	**0.31479**	**0.17335**	**0.13219**
da66efed Pearson	10	0.57730	0.40708	0.54254
e211303d Pearson	**655**	**0.61370**	**0.49447**	**0.31414**
e26c6647 Pearson	12	0.23264	-0.03143	0.10237
e492e34c Pearson	18	0.26888	0.01146	0.13832
e6dca229 Pearson	3	0.67109	0.59604	0.84543
efcc2179 Pearson	48	0.77561	0.32192	0.40781
fdddf728 Pearson	7	0.13692	0.01429	0.00809
		Diff Delta	**Commit Count**	**Lines of Code Changed**
Pearson Correlation for Dataset (weighted average)		**0.3834**	**0.2702**	**0.2503**
Max Correlation (min n=100)		**0.6137**	**0.4945**	**0.3141**
Diff Delta vs Others		**41.88%**	**-29.52%**	**-34.71%**
Data points (n)	**2729**			
Data points (repos)	**61**			

R2 correlation

Repo correlation coefficients	Record count	Diff Delta	Commit Count	Lines of Code Changed
04bffefd R2	6	0.101998	0.029486	0.163987
07a6a2fa R2	3	0.047689	0.395161	0.211947
097b4331 R2	12	0.274220	0.015707	0.031227
0a6499bf R2	**299**	**0.067022**	**0.067267**	**0.049298**
0c492158 R2	26	0.049967	0.006833	0.003345
0f2aaf30 R2	12	0.055277	0.001320	0.028864
1353a4c4 R2	24	0.000713	0.016011	0.017936
17730c68 R2	8	0.006108	0.001552	0.019792
1d88ad13 R2	40	0.042542	0.000311	0.000213
2b682982 R2	11	0.647068	0.187133	0.649837
2cfcbf23 R2	27	0.138998	0.115722	0.114299
2fadc9df R2	16	0.135367	0.012277	0.089817
30a64ca0 R2	10	0.458907	0.150531	0.091267
3242485d R2	8	0.095338	0.125749	0.142243
3a218db3 R2	46	0.454449	0.222012	0.199769
3c8d9e25 R2	11	0.486838	0.723173	0.775347
3ed5dcff R2	44	0.026772	0.001697	0.003235
472bc4dd R2	13	0.252524	0.531215	0.453802
4acd4acf R2	3	0.027418	0.024194	0.205600
4d69f0ac R2	22	0.059919	0.103196	0.037375
50315b0e R2	5	0.962651	0.986845	0.998795
50a0767b R2	48	0.026431	0.156856	0.012850
5a9ebe3c R2	4	0.758439	0.235294	0.874317
5ffac53b R2	19	0.036905	0.001026	0.042687
606dfa8b R2	9	0.083601	0.037692	0.005296
66779d5f R2	5	0.597407	0.463300	0.588483
674f7b2d R2	9	0.086000	0.151405	0.201676
72a585e3 R2	8	0.305120	0.082824	0.116386
752b1dd0 R2	30	0.052137	0.003637	0.045435

75305ee2 R2	10	0.068112	0.080601	0.072920
7e1f5915 R2	54	0.087871	0.000003	0.035564
7e9398b4 R2	8	0.022203	0.013083	0.002075
7ef3d268 R2	12	0.529788	0.165497	0.338162
80abc79a R2	3	0.203807	0.644737	0.249036
80c6eedd R2	4	0.369736	0.367385	0.668798
94aefe51 R2	15	0.116264	0.003762	0.035762
9a7f05e0 R2	4	0.345012	0.081342	0.054898
9b4ceb2f R2	3	0.577063	0.285637	0.404626
a90b4fc9 R2	6	0.060641	0.030488	0.044239
ab5f83c9 R2	4	0.150064	0.049587	0.007549
acf55047 R2	20	0.349633	0.471869	0.260519
b5078827 R2	12	0.106486	0.133994	0.048229
b6d9794a R2	13	0.020204	0.041883	0.000223
b6fc5e31 R2	**427**	**0.070202**	**0.034347**	**0.084964**
b7260273 R2	9	0.011932	0.001619	0.026062
bcc4da08 R2	5	0.077972	0.112252	0.159928
c08ca054 R2	10	0.147438	0.246985	0.021424
c3ea74b0 R2	**178**	**0.151263**	**0.084084**	**0.022927**
c4abd096 R2	12	0.245597	0.007600	0.403701
c556a08a R2	48	0.013850	0.032323	0.008336
caa66f10 R2	26	0.199054	0.018440	0.119772
cf472176 R2	17	0.229691	0.095125	0.166140
d231b61d R2	7	0.191249	0.240536	0.256241
d4005b42 R2	3	0.998215	0.998521	0.998912
d6041c05 R2	**288**	**0.099095**	**0.030052**	**0.017473**
da66efed R2	10	0.333276	0.165819	0.294354
e211303d R2	**655**	**0.376630**	**0.244498**	**0.098682**
e26c6647 R2	12	0.054123	0.000988	0.010479
e492e34c R2	18	0.072294	0.000131	0.019134
e6dca229 R2	3	0.450366	0.355263	0.714752
efcc2179 R2	48	0.601575	0.103636	0.166308
fdddf728 R2	7	0.018748	0.000204	0.000066
		Diff Delta	**Commit Count**	**Lines of Code Changed**

Correlation Coefficient for Dataset (weighted average)		**0.1876**	**0.1147**	**0.0853**
Max Correlation (min n=100)		**0.3766**	**0.2445**	**0.0987**
Diff Delta vs Others		**63.53%**	**-38.85%**	**-54.54%**
Data points	**2729**			

Responsible transparency disclosure

Because GitClear has been publishing articles on the relationship between code output and cognitive energy since 2018, this paper cites GitClear data and links throughout. As mentioned in the Results section, the dataset is provided by GitClear, using anonymized git stats from companies who opted into sharing industry data. The author of this paper, William Bates Harding, uses the title “Programmer/CEO of Alloy.dev” to describe his work on LinkedIn. Alloy.dev has a website available at https://alloy.dev, where GitClear is listed among the three products that the company operates, along with Amplenote and Bonanza.com.

GitClear has had an open job posting to hire a professional researcher to analyze its data since a blog post made three months ago, in November 2020. The paid research opportunity was circulated via email among academic researchers prior to the job being posted on the GitClear blog. Which is to say, prolonged efforts have been made to contract an existing Professional Researcher on a paid study that leverages the GitClear dataset. Because GitClear has yet to connect with such a Researcher as of February 2021, the team decided to publish this first analysis of its own dataset. Assuming this paper sparks greater interest in software metrics, it’s expected that follow-up research could be undertaken by additional third parties.

By publishing the source code that was used to generate this dataset, we believe the data produced in this study will be reproducible by other parties that independently undertake an evaluation using similar parameters. We will assist any such efforts to the extent we are able. If this is a topic that is interesting to you, please reach out to the email address bill at gitclear.com and we can talk about how to advance further research in the software metrics space.

The repo “e211303d” cited in the data corresponds to the source code for Bonanza.com, a repo that is operated by Alloy.dev. Among the n=655 issues with Story Points for Bonanza, here is the breakdown of when the tasks were estimated and worked on:

- 2017: 140 issues
- 2018: 302 issues
- 2019: 169 issues
- 2020: 71 issues
- 2021: 7 issues

Story Points were never altered after being initially chosen: once an estimate was made, it was left as a record for future evaluation. Thus, this dataset is considered as valid as any other since it was accumulated well in advance of the decision to undertake this research (save for some of the 7 issues from 2021. The internal effort to collect and research this dataset began in January 2021).

It is reasonable conjecture that two factors may contribute to the high correlation levels observed by the Bonanza repo:

1. Developers making Sprint Point estimates without input from external stakeholders. While some like Atlassian recommend bringing in stakeholders across disciplines to estimate Story Points, the presence of non-technical members participating in the "effort analysis" process seems likely to reduce the extent to which a developer's assessment of the technical complexity is utilized as the final source of truth to determine Story Points.
2. Since the Bonanza repo was the first repo analyzed by GitClear, it was vetted to ensure that auto-generated/third-party directories and files were not included in its source code analysis. These steps also ensure that no Lines of Code will be accumulated by extraneous sources, so the correlation for Lines of Code Changed is higher than it would be otherwise.

As GitClear is adopted by more teams, it's expected that Bonanza's correlation levels will be eclipsed by other startup teams with low tech debt, that endeavor to choose Story Points based solely on their technical complexity. Already, there are 12 small repos in the 61 repo dataset with r^2 levels higher than Bonanza's 38%.

Full Source Code Used for Research

At this link, find the full data set showing correlation between Story Points and developer metrics on a per-repo basis. The following Ruby source code was used to generate the CSV file:

```ruby
# ------------------------------------------------------------------------
# Job to compile stats that establish r^2 between Diff Delta and Story Points
class GenerateStoryPointCorrelationCsv < ApplicationJob
 MIN_RECORDS_PER_REPO = 20

 INDIVIDUAL_CSV_COLUMN_ORDER = [
   :repo_identifier,
   :story_point,
   :line_impact,
   :commit_count,
   :loc_changed,
 ]

 AGGREGATE_CSV_COLUMN_ORDER = [
   :entity_identifier,
```

```
    :repo_identifier,
    :story_point,
    :story_point_record_count,
    :impact_per_issue,
    :commit_per_issue,
    :loc_per_issue,
  ]

  # ------------------------------------------------------------------------
  def perform
    now = Time.now
    freshen_story_point_research_stats!

    story_point_scope = StoryPointResearchStat.where("story_points >
0").joins(:repo_issue).
      where("story_point_research_stats.line_impact > 0 AND
story_point_research_stats.commit_count > 0")

    story_point_scope = story_point_scope.where(updated_at: 1.day.ago..Time.current)

    generate_detailed_stats!(story_point_scope)
    logger.info "Completed stat generation in #{ Time.now - now }"
  end

  # ------------------------------------------------------------------------
  def freshen_story_point_research_stats!
    processed = 0
    StoryPointResearchStat.joins(:repo_issue).find_each do |research_stat|
      research_stat.populate_from_issue!(research_stat.repo_issue)
      processed += 1
      logger.info("Processed #{ processed } SPRS records, up to ##{ research_stat.id
}") if processed % 100 == 0
    end
  end

  # ------------------------------------------------------------------------
  def generate_detailed_stats!(story_point_scope)
    file_destination ||= "#{ Rails.root }/story_point_correlation.csv"
    story_point_scope = story_point_scope.order(:repo_uniqueness_md5, :story_points)
    impact_tuples, commit_tuples, loc_tuples, repo_rows, all_pearsons, all_r2s = [],
[], [], [], [], []

    CSV.open(file_destination, "wb") do |csv|
      csv << INDIVIDUAL_CSV_COLUMN_ORDER

      story_point_records = story_point_scope.to_a
      story_point_records.each_with_index do |research_stat, index|
        next unless (repo_identity = research_stat.repo_uniqueness_md5)

        impact_tuples << [ research_stat.story_points, research_stat.line_impact ]
        commit_tuples << [ research_stat.story_points, research_stat.commit_count ]
        loc_tuples << [ research_stat.story_points, research_stat.lines_changed ]

        repo_rows << [
          repo_identity, # :repo_identifier,
```

```
          research_stat.story_points, # :story_point,
          research_stat.line_impact, # :line_impact
          research_stat.commit_count, # :commit_count,
          research_stat.lines_changed, # :loc_changed,
        ]

        if repo_rows.present? && repo_identity != story_point_records[index +
1]&.repo_uniqueness_md5
          impact_correlation, commit_correlation, loc_correlation =
            MathUtility.pearson(impact_tuples), MathUtility.pearson(commit_tuples),
MathUtility.pearson(loc_tuples)

          unless impact_correlation && commit_correlation && loc_correlation
            logger.error "Got a missing correlation among [#{ impact_correlation }, #{
commit_correlation }, #{ loc_correlation }] while processing #{ repo_rows.inspect }"
            next
          end

          if repo_rows.size > 2 && (impact_correlation > 0 || commit_correlation > 0 ||
loc_correlation > 0)
            repo_rows.each do |csv_row|
              csv << csv_row
            end

            pearsons = [ "#{ repo_identity } Pearson", repo_rows.size,
impact_correlation,
              commit_correlation, loc_correlation ]
            all_pearsons << pearsons
            r2s = [ "#{ repo_identity } R2", repo_rows.size,
MathUtility.r2(impact_tuples),
              MathUtility.r2(commit_tuples), MathUtility.r2(loc_tuples) ]
            all_r2s << r2s
            csv << [""]
          end

          impact_tuples, commit_tuples, loc_tuples, repo_rows = [], [], [], []
        end
      end

      csv << [""]
      csv << ["Repo pearson correlations", "Record count"]
      all_pearsons.each do |pearson_row|
        csv << pearson_row
      end

      csv << [""]
      csv << [""]

      csv << ["Repo correlation coefficients", "Record count"]
      all_r2s.each do |r2_row|
        csv << r2_row
      end

      csv << [""]
    end
```

```
 end
end
```

The implementation references two methods in MathUtility. Here is the method used to calculate Pearson correlation:

```
# ------------------------------------------------------------------------
# Pearson correlation, aka the "r" in "r^2". How closely do a dependent and
independent variable
# correlate with one another?
def pearson(data_tuples)
 return nil unless data_tuples.present?

 sum_independent = data_tuples.map(&:first).sum
 sum_dependent = data_tuples.map(&:last).sum

 sum_independent_squares = data_tuples.map(&:first).sum { |x| x**2 }
 sum_dependent_squares = data_tuples.map(&:last).sum { |y| y**2 }

 products = data_tuples.inject([]) do |arr, (x, y)|
   arr << x * y
 end

 # Calculate Pearson score
 numerator = products.sum - (sum_independent.to_f * sum_dependent /
data_tuples.size)
 denominator = ((sum_independent_squares - (sum_independent**2) /
data_tuples.size) *
   (sum_dependent_squares - (sum_dependent**2) / data_tuples.size))**0.5

 return nil if denominator == 0
 numerator / denominator
end
```

And here is the method used to calculate r^2 correlation:

```
# ------------------------------------------------------------------------
def r2(data_tuples, logger: nil)
 slope = MathUtility.regression_slope(data_tuples)
 y_intercept = MathUtility.y_intercept(data_tuples, slope)
 y_mean = data_tuples.map(&:last).mean

 # This is a useful graph to visualize what these two quantities represent
https://www.graphpad.com/guides/prism/8/curve-fitting/images/_bm36.png
 sum_of_squared_error = data_tuples.sum(0) do |x, y|
   ((y_intercept + slope * x) - y) ** 2
 end
```

```
 sum_of_squares = data_tuples.sum(0) do |_x, y|
   (y_mean - y) ** 2
 end

 if logger
   logger.info "Analyzed #{ data_tuples.size } data tuples, found y_intercept
#{ y_intercept }, slope #{ slope } and mean #{ y_mean }. Equates to
sum_of_squared_error #{ sum_of_squared_error } and sum_of_squares #{
sum_of_squares }"
 end

 1 - (sum_of_squared_error / sum_of_squares)
end
```

Additionally, the method makes use of a StoryPointResearchStat model from which the data is drawn. Here is the database definition underlying that model:

```
# Table name: story_point_research_stats
#
#  id                     :bigint           not null, primary key
#  story_points           :decimal(8, 1)    not null
#  line_impact            :integer          not null
#  commit_count           :integer          not null
#  lines_changed          :integer          not null
#  entity_uniqueness_md5  :string(8)        not null
#  created_at             :datetime         not null
#  updated_at             :datetime         not null
#  repo_uniqueness_md5    :string(8)
```

Here is the job that generates the data for that model:

```
class GenerateStoryPointResearchStat < ApplicationJob
 # ------------------------------------------------------------------------
 def perform(entity)
   if entity.settings.shares_industry_stats
     issue_scope = RepoIssue.joins(:extra, :repo).where(repos: { entity_id: entity.id
}).left_joins(:research_stat).
       where("repo_issue_extras.story_points > 0").where(story_point_research_stats: {
id: nil }).resolved.where("closed_at < ?", 3.days.ago)

     if issue_scope.exists?
       logger.info "Found story point research records to generate"
       issue_scope.each do |repo_issue|
         next unless repo_issue.commits.exists?
         StoryPointResearchStat.generate_for_issue(repo_issue)
         logger.info "Generated research stats for issue ##{ repo_issue.id }"
       end
     end

     logger.info "Finished creating #{ issue_scope.count } StoryPointResearchStats"
   else
```

```
      logger.error "Error: Entity ##{ entity.id } does not share industry stats"
    end
  end
end
```

It creates a StoryPointResearchStat record for any entity (i.e., company) that shares industry stats, and has Jira issues (aka RepoIssue) that possess non-nil Story Points. Here is the class definition for StoryPointResearchStat itself, which is referenced in the job above.

```
class StoryPointResearchStat < ApplicationRecord
 # ------------------------------------------------------------------------------
 def self.generate_for_issue(issue)
   if (existing_stat = StoryPointResearchStat.where(repo_issue_id: issue.id).first)
     existing_stat.populate_from_issue!(issue)
   else
     StoryPointResearchStat.new.populate_from_issue!(issue)
   end
 end

 # ------------------------------------------------------------------------------
 def populate_from_issue!(issue)
   issue_impact = issue.commits.impacting.sum(:value)
   self.commit_count = issue.commits.count
   self.entity_uniqueness_md5 = issue.entity.extra.derive_uniqueness_md5
   self.line_impact = issue_impact
   self.lines_changed = CodeLine.changed.where(commit_id:
issue.commits.select(:id)).count
   self.repo_issue_id = issue.id
   self.story_points = issue.story_points
   self.repo_uniqueness_md5 = issue.repo.extra.derive_uniqueness_md5
   self.updated_at = Time.current # Ensure we update timestamp even if no value
changed

   if story_points
     save!
   else
     destroy
   end
  end
end
```

We welcome suggestions to improve the design of any of this source code, or this study as a whole. We intend to sponsor future analyses of the dataset as it grows.