Benchmarking ongoing development output in real-life software projects

Jonathan Streit[0009−0006−8129−3184] and Lukas Feye

itestra GmbH, Destouchesstr. 68, 80796 Munich, Germany
streit@itestra.de

Abstract. In this case study we compare six different metrics and their suitability for productivity benchmarking on development output level. We use detailed data from four software industry projects performed by one company with overall 264 months of development and 1.1 million source lines of code. Code change, absolute growth and number of commits as well as invested effort are measured in consecutive 3-month periods. This allows us to observe alterations in productivity throughout the course of a project as well as inter-project comparisons. We find correlations between effort and the chosen output metrics as well as significant and explainable productivity differences between projects and project phases.
We also analyze whether the use of a clone detection algorithm can improve measurement by adjusting for copy & paste additions and renamed or moved code, and find that a small benefit exists. The redundancy-adjusted amount of code tokens added or modified seems to be the best metric among the selected, in particular in ongoing development where an already existing codebase is changed. Number of commits and absolute growth may complement the picture.

Keywords: Metrics · Productivity · Empiric Case Study

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

1 Introduction

Being able to measure and benchmark software development output and in consequence productivity is crucial for both project-internal steering as well as for management of an IT organization. However, it is also a difficult task for which the single silver bullet metric does not exist [10]. Beck describes in [4] the four levels of *effort*, *output*, *outcome* and *impact* on which measurement can take place[1]. While it seems desirable to measure "high level" aspects such as economic value of a feature, this is at the same time impracticable for many applications due

[1] with *outcome* describing the value for the customer and *impact* the value that flows back to the IT organization

to the required measurement effort, the longer feedback cycle and the numerous factors outside the IT organization that can influence the result. We therefore believe that output measures, such as code size-related metrics, will remain an important tool.

In order to evaluate productivity beyond relative comparisons, a benchmark scale is required. Such an absolute scale allows to asses the status of a project or development team and to estimate the potential of improvement actions. We do this in our consulting projects, so-called software HealthChecks [19], where modernisation and optimization strategies are developed including ROI estimates. Benchmarks can also be helpful when a project starts and historic data is not yet available for comparison.

In this case study we analyze the suitability of different output metrics for productivity benchmarking using four real-world industry projects. Main contributions compared to the state of research are:

1. We perform a fine-grained analysis of real-world software industry projects, including data on invested effort. Existing research often uses aggregated total values or open source projects, for which no effort data exists.
2. We compare metrics for their suitability to measure change within an existing codebase. Working on an existing system is the predominant situation in practice. Popular metrics and data in benchmark collections are however often limited to sizing absolute amounts of software that have been constructed (from scratch) for a certain amount of money or effort.
3. We analyze whether the use of a clone detection algorithm to account for duplicated, renamed or moved code can improve measurements.

2 Background and related work

Productivity measurement. Both practitioners and researchers have worked on productivity metrics for decades. Beck [4] and Forsgren [10] discuss different levels on which output and productivity measurement may take place. According to Oliveira [17], lines of code per effort or cost and function points per effort or cost are the most frequently used in research. Lines of code are easy and cheap to measure, but at the same time criticized for being potentially misleading [2]. Due to their popularity in practice, we will focus this case study on code-based metrics such as changed lines of code. Function points aim to abstract from concrete implementation technologies. They were initially designed for manual counting during effort estimation (i.e. before code is available). In the context of agile projects, "velocity" metrics such as story points [5] have gained popularity, but they can only be used for a relative, project-internal comparison as no absolute benchmark scale exists and each team has their own notion of how much effort and/or result a story point represents.

Measuring code change. Lines of code, function points and many other metrics are in the first instance measures for absolute size, not for code change. Explicit change measures, also called "code-churn" metrics, have been used intensely in research for defect density prediction and, building on top of that,

prioritization of test cases and review efforts. Usually the number of added or modified lines in a module or the number of commits is used, e.g. in [16] and [21]. Some works in the area of software evolution such as [8] also use change measures for correlations with quality metrics. Fewer works have analyzed explicitly code change in the context of development output and productivity measurement. Kocaguneli [14] researches the relation between code churn and developer assignment and finds that the planned work distribution between the developers is not reflected by the actual distribution of code change. Ramil and Lehman [9] analyze code change metrics on different levels of granularity as effort predictors.

Industry data. While measurement of productivity has been an intensely discussed topic in both academia and industry for many years, there are only few works that provide fine-grained analyses of industry projects and actual effort data. [22] is such an example, analyzing output and effort for six builds of one project. There is no description in the paper how exactly the change from one build to the next is measured, or whether only the absolute growth is counted. Kocaguneli [14] analyzes several industry projects, but does not consider effort data. Ramil and Lehman [9] report on an industrial mainframe software, but the effort data used is only derived from change log entries.

Benchmark collections such as in [11, 1] provide productivity values for a variety of domains, technologies etc. but only as aggregated total values, without insights on the course of the projects or further characteristics. Barb [3] uses data from [1] to analyze, amongst others, whether lines of code can predict invested effort and concludes that no. We however argue that productivity varies significantly between projects and thus a linear relation between lines of code and effort across projects cannot necessarily be expected.

Redundancy-free size measurement. Mas and Pizka [18] have suggested to use clone detection in order to improve shortcomings of pure lines of code measurements and established the *R*edundancy-*F*ree *SLoC* metric.

3 Methodology and data

For our study, we formulate the following research questions:

- **RQ1** Can code change related metrics provide insights on development productivity?
- **RQ2** Which of the metrics is most suitable for productivity benchmarking?
- **RQ3** Can the use of clone detection improve measurements?

3.1 Metrics

We use six metrics to measure development output during a given time period:

- *Changed LoC.* The number of lines added or modified in the diff [15] between the code version at the beginning of the time period and the version at the end of the time period. Deleted lines, changes in whitespace and empty lines are not counted. Changes in comment lines are included in this metric.

- *RFCC.* The *R*edundancy-*F*ree *C*ode *C*hurn metric (*RFCC*) has been developed at our company itestra and used for around 15 years. It measures the amount of code added or modified between the code version at the beginning of the time period and the version at the end of the time period, adjusted for code that has just been moved around or duplicated. A clone detection algorithm is used to identify such parts. Details of the algorithm are presented in the next section. In order to be independent from the formatting style, this measurement is based upon source code tokens rather than physical lines. Comments and `import`/`using`-Statements are not counted.
- *CC* (*C*ode *C*hurn). In order to judge the impact of the redundancy-adjustment of *RFCC*, this helper metric is calculated with exactly the same algorithm as *RFCC* but without the clone detection and adjustment part.
- *RFSLoC absolute growth.* The *R*edundancy-*F*ree *SLoC* metric as defined in [18] measures the size of a codebase in source lines relieved of redundancies from code cloning. *RFSLoC absolute growth* is defined as the redundancy-free size at the end of the time period minus the redundancy-free-size at the beginning. Note that, differing from the above metrics, this sums up added and deleted lines and may yield negative results when large amounts of code were deleted. The tool ConQAT [7, 13] is used for the analysis. Comments and `import`/`using`-Statements are not counted, nor are modified lines.
- *SLoC absolute growth* is defined as the number of code lines at the end of the time period minus the number of code lines at the beginning. The counting is performed analogously to the *RFSLoC* metric, i.e. comments and `import`/`using`-Statements are not counted.
- *Commits.* The number of commits in the version control system during the time period. Different from the other five, this metric is not related to the actual program code.

From each output metric a corresponding productivity metric can be calculated by dividing the output by the effort invested in the same time period.

Test code, generated code and source code of 3rd party libraries are excluded manually from the code base before measurement.

3.2 The RFCC algorithm

The *RFCC* calculation is performed as follows:

1. For each file in the codebase, both the code version at the beginning of the time period and the version at the end of the time period are transformed into token sequences using the lexer from ConQAT [7, 13]. Files that are present in only one of the versions are treated as empty in the other.
2. Comments and `import`/`using`-Statements are filtered out.
3. The diff [15] between the two token sequences is calculated for each file. Deletions are discarded, i.e. we keep added and modified tokens (the latter are represented as an addition and a deletion in the diff output).

4. Clone detection is performed on the union of all token sequences from the beginning of the time period and the diff results. The ConQAT framework and its clone detection component are used as a base for the implementation. Clones are required to have a minimum length of 50 tokens. Sequences differing only in literal values are still recognized as clones.
5. Tokens from the diff that appear in any clone also spanning a part of the code version at the beginning of the time period are marked as not to be counted in step (7). This has the effect that code that was moved to another location is not counted. The rationale is that moving code or renaming a package or module is a simple operation that would appear as an unjustified large change in a naive diff.
6. Tokens from the diff that appear in clones only spanning the diff part are counted with $\frac{1}{n}$ in step (7) with n being the number of clone instances. This has the effect that code that was added multiple times is counted exactly once, similar as in *RFSLoC* [18]. The rationale is that code clones are considered an anti-pattern regarding maintainability and that adding redundant code several times does not increase functionality n-fold.
7. As a last step, the tokens from the diff are counted. The result is transformed back into a lines of code-like scale by dividing through 4.6 as an average number of tokens per source line for Java and alike.

3.3 Projects in the study

Four software projects performed by the company itestra for clients in the financial domain between 2015 and 2023 were analyzed for this study. All projects dealt with individually developed business information systems and used Java or C# as their main programming language. Only projects with a duration of several years were selected.

- Project A created an integrated insurance system and spans two phases. During the *development phase* which lasted approximately $3\frac{1}{4}$ years, the system was constructed using both a legacy system and additional requirements as specification. The code was put in production incrementally. In the following *maintenance phase* functional changes, enhancements and maintenance tasks were performed.
- Project B migrated a payroll backend system from COBOL to C#, preserving the functionality and exact result but with a completely new technical design. Testing intensified in the second half of the project and there was only one "go live" date towards the end.
- Project C used an existing integrated insurance system as a starting point and adapted and extended it for a new customer.
- Project D developed a system that simulates an outdated insurance system to maintain data legally mandated to be retained. The project subdivides into a *development* and a subsequent extension and *maintenance phase*.

Effort data was extracted from time bookings in itestra's ERP system and comprises the whole development team and activities including design, coding,

developer testing, bugfixing, meetings and itestra-side project management. It does not include acceptance testing, requirements specification and customer-side project management. Effort is in person days (8h) unless noted otherwise. Code versions for the selected points in time were obtained from the main development branch of the version control system for each project.

The overall observation periods used for the study were chosen as large as possible, starting after project initialization[2] and ending e.g. when the customer took over in maintenance and thus a complete retrieval of effort data was not possible any more. Table 1 provides some key figures on the size of the projects.

Table 1. Project overview. Code size is measured in SLoC. Legacy systems serving as reference are *not* included in the code size.

Id	Code size at start[2]	Code size at end	Months duration	Total person days	Total commits
A	66,248	398,236	96	6,559	62,476
B	13,833	397,917	42	4,912	23,195
C	268,685	349,530	60	4,004	22,848
D	14,531	55,342	66	1,084	4,218

3.4 Evaluation methods

As we are particularly interested in measuring ongoing software development, i.e. work in an existing code base, we subdivide each project into consecutive time periods of constant length. We can thus observe both alterations throughout the course of a project and inter-project differences. We expect from a meaningful output metric that

- it should correlate with the invested effort within the scope of one project and homogeneous project phase, i.e. productivity should not vary strongly.
- it should yield differences in values for different projects and different project phases that can be explained by the context.
- it should not be prone to systematic errors or easy to manipulate.
- it should have an understandable interrelation with the output of software development, i.e. implementing or changing features of a software system.

We will thus for the evaluation of RQ1 and RQ2

- calculate correlation coefficients and their statistical significance for all pairs of an output metric and the effort invested in this time period for each project separately. Suitable metrics should have a high correlation with effort within a project phase.
- calculate the mean productivity values for each project phase and metric as well as their variance resp. standard error.

[2] Projects A, B and D do not start at zero due to sample and prototype code that was created before the actual project start.

- compare mean productivity values between projects and project phases and analyze the statistical significance of the differences using a 2-sided t-test. Suitable metrics should show some (explainable) variation between projects and project phases.
- analyze possibilities of systematic errors and manipulations.

Besides, repeatable and objective results as well as low measurement effort are desirable, which all metrics used here fulfill as they are automated measures.

Note that we aim to measure development output, not business value, and thus do not attempt to relate metric results to any sizing on business level.

For RQ3 we will compare the above results for *RFCC* vs. *CC* and *RFSLoC absolute growth* vs. *SLoC absolute growth*. Additionally we will analyze a sample of changes on code level for time periods where *RFCC* and *CC* deviate.

3.5 Division into time periods

Overall observation time for each project was split into consecutive time periods of 3 months. We chose this length as it is on one hand small enough to generate a sufficient number of data points and on the other hand large enough to equalize effects like daily variation, people working locally for a few days before committing, corrections of a recent check-in etc.

4 Results

Figure 1 displays the different output metrics as well as the invested effort throughout the course of the four projects. Each data point represents a consecutive 3-month period. In order to keep the diagram readable, *CC* and *SLoC absolute growth* have been omitted from this and other figures.

Figure 2 shows the Spearman correlation coefficients for the correlation between each output metric and effort for each of the projects. The correlation is statistically significant except for *RFSLoC absolute growth* and *SLoC absolute growth* in project C which is plausible as the project includes periods when large amounts of code, representing functionality from the imported system that was not needed by this customer, were removed from the codebase and thus the absolute growth was low or even negative. Correlation coefficients—except for the two data points in project C—range from 0.67 to 0.88. On average they are highest for *Commits*, then *RFCC* followed closely by *Changed LoC* and *CC* and lowest for *RFSLoC absolute growth* and *SLoC absolute growth*. We would have expected the last two metrics, which only take into account absolute growth, to show little correlation with effort during maintenance/bugfixing, but in the maintenance phase of A and the second half of B the correlation is also significant.

Figures 3 to 6 show the mean productivity values for each metric for the different projects and project phases and table 2 displays t-test results for selected pairs.

In project A, we can see a strong and statistically significant drop in *RFCC per person day* from the development phase to the maintenance phase (similar for the other code-based metrics), while at the same time *Commits per person day* increase slightly. This could be explained by different characteristics of the project work, i.e. smaller changes with higher analysis and communication effort. Productivity could however also have been influenced by changes in the team structure: during the development phase only six developers contributed more than 90% of the commits, at least two of them seniors. In the following $3\frac{1}{4}$ years the same share of commits was spread over 17 developers and only two of them had participated significantly in the development phase, indicating a loss of trained team members.

There is also a statistically significant drop in *RFCC per person day* (and similarly for the other code-based metrics) in project B when comparing the first and the second half of the project while *Commits per person day* remains constant. This could be explained by the fact that testing and thus also bugfixing was performed mostly in the second half, causing more commits with only few lines changed.

Table 2. t-test results for mean productivity value comparison between projects and project phases. See section 3.3 for an explanation of the project phases.

p-values for	vs.	RFCC p. PD	Commits p. PD
A dev	A maint.	**0.01**	0.95
B first half	B second half	**0.000**	0.67
D dev	D maint.	0.41	0.27
A dev	B first half	0.77	**0.000**
A dev	C	0.11	**0.000**
A dev	D dev.	0.07	**0.000**
B first half	C	0.09	**0.02**
B first half	D dev	0.07	0.95
C	D dev	0.45	0.11

Commits per person day are on a similar level in all projects and project phases except for project A where they are about twice as high as in the other projects. This difference is statistically significant. The only possible cause we see would be a different team culture where developers try to integrate their changes as soon as possible.

The other metrics show moderate inter-project variation. Average *RFCC per person day* is between 75 and 111, which is higher than the average of 50 we have observed in comparable industry projects. For the low value of *RFSLoC absolute growth* in project C see the explanation above.

5 Discussion

Our data shows correlations between the effort invested and any of the chosen output metrics as well as significant differences in productivity between projects and project phases. We consider this an indicator for the expressiveness of the metrics, as described in section 3.4. There are however particular strengths and weaknesses of each metric that we will discuss in the following.

First of all, *Commits per person day* differs from the other metrics. In project A and B, it changes little with the transition from development to maintenance/bugfixing phases (see figure 6). This could be seen both as an advantage—making work comparable regardless of the project phase—and as a drawback because the fact that less software is created for the same effort is not reflected. *Commits* can be easily counted for arbitrary time periods, but the result is obviously influenced by the development process and branching strategy used, hindering inter-project comparison. For instance, merging the trunk into a feature branch to keep it up to date or committing small changes to see the results in the CI pipeline will increase the number of commits, while squashing commits in Git or working locally before committing a larger feature will reduce it. The metric could also be easily manipulated by committing more often, and frequent corrections could lead to a higher commit number although it is always the same code that is changed.

Changed LoC, *RFCC* and *CC*, i.e. the metrics that measure actual code change, are inherently vulnerable to an effect known as *coastline paradox* in other domains [20]. The smaller the measurement interval is chosen, the larger the total sum for the length of an irregular path like a coastline will be. In our case, the smaller the length of the time periods to be analyzed is chosen, the larger the sum of all change measures for the overall project. For instance, a code line that is changed or corrected multiple times will be recognized as a single change with a large interval while counting multiple times with smaller intervals[3]. Similarly, code that is temporarily added and later removed may not appear with a large analysis interval at all[4]. In order to counter this effect, the analysis interval should neither be chosen too small nor too large, from our experience between 3 and 12 months. In order to quantify the effect, we experimentally varied the interval length for project C: using 1 / 2 / 6 / 12 months instead of 3 changes the total sum by +20% / +4% / −4% / −10%.

The same metrics are also vulnerable to automated refactorings that change a lot of code with a single command, such as renaming a package. *Changed LoC* is most vulnerable as it is even affected by changes in code format, *RFCC* least as it will detect and ignore certain refactorings such as moving code around.

In contrast, *SLoC absolute growth* and *RFSLoC absolute growth* are immune to these effects but can, on the other hand, not detect deliberate changes of

[3] The extreme case would be to analyze every commit separately, as in the Git "Lines of code changed" statistics.

[4] From a business point of view, changing the code multiple times does not represent economic value unless it was for a temporarily needed feature.

existing functionality or a situation where large amounts of code were deleted and new code added, as in project C. This is a drawback for ongoing development projects.

All code-based metrics can be influenced by adding generated code or 3rd party modules to the codebase, which is why we demand to exclude such code from the measurement (see section 3.1). On the other hand, the code-based metrics are closer to the output of software development, i.e. implementing or changing features of a software system, than *Commits*. *RFCC* and *RFSLoC absolute growth* are less vulnerable to accidentally included generated code as this code is usually highly redundant.

Conclusively, we answer **RQ1** positively. The measurement results fit with the actual project histories, e.g. phase transitions, and we therefore think the metrics can provide insights on development productivity also in other projects and allow for inter-project comparison. Automation allows for repeatable, low-cost and thus continuous measurement. The metrics should however be used with caution, as discussed in the following section.

No single metric stands out for **RQ2**: *RFCC* has slightly higher correlation than the other code-based metrics but for the ease of measurement any other code change metric could also be employed. *Commits* can be used for comparisons within a project but is not well suited for inter-project comparison or benchmarking. It could be used as a complement that provides a different perspective. Similarly, we consider absolute growth metrics such as *RFSLoC* a possible complement.

RQ3: Correlations are slightly higher and more significant for the redundancy-adjusted versions of the metrics (*RFCC* and *RFSLoC absolute growth*) than for the normal ones (*CC* and *SLoC absolute growth*). In order to verify this effect from another perspective, we analyzed in detail the five time periods of project A where *RFCC* and *CC* differ significantly (see figure 7 in red). Two of them represent the starting phase, the three others contained large changes that renamed, split or moved modules, i.e. situations where considerable amounts of code were affected with moderate effort. The lower value of *RFCC* compared to *CC* seems more plausible here.

We can thus answer this RQ positively, although the difference is small. As the company in the study attaches great importance to clone avoidance, the difference may be larger in the average software project than in the ones here. A drawback of the redundancy-adjusted metrics is that more advanced measurement tooling is needed and the results are not as straightforward to comprehend and communicate.

5.1 Use with caution

The use of code-size related metrics for benchmarking bears well-known risks as discussed in [18]. Depending on its use, the information may be interpreted wrongly or create incentives to write code as verbose as possible. We therefore strongly recommend to

- not use any of the metrics presented here as the only source of information. They should be complemented e.g. by tracking completion of planned features, customer satisfaction, defect rates or code quality. Discrepancies can provide valuable insights, e.g. high activity in the code but slow progress from a feature perspective may indicate architectural overhead.
- not measure too finely, such as by week or individual developer. Longer time periods will level out noise from daily variation, the coastline paradox discussed above and effects such as someone writing code in a "quick and dirty" manner and having to correct it afterwards, possibly in multiple attempts. Team scope reduces the incentive to adapt to the metric and accounts for heterogeneous role distributions such as one person doing all reviews.
- include most project activities into effort calculation, such as developer meetings and ceremonies, technical design, code reviews and bug fixing. This puts the focus on overall productivity and appreciates e.g. time spent in thorough design to avoid rework. Activities related to requirements engineering, acceptance testing, operation and support can be excluded from effort data.
- be consistent about inclusion and exclusion of code parts and effort.
- not attach penalties or incentives for the developers directly to the measurements as this will make people adapt their working style to the metric, e.g. committing frequently or writing particularly verbose code. As discussed in [18] the redundancy-free metrics seem less vulnerable to manipulation. In the present study measurement was performed ex post so no incentive to manipulate the metrics existed for the developers.
- consider the context when comparing between projects. This should not be used as an excuse (most people consider the system they are working on particularly "complex" and "critical") but of course working on a large taxation calculation core will require more analysis effort than building a simple database frontend. Expressiveness of programming language differs (see [12] for lines of code per function point tables) as well as available tool support and libraries e.g. for Java and COBOL.

5.2 Threats to validity

The study was performed with only four projects from the same company and similar domains and technology stacks, which may limit generalizability.

Separation of project phases, attribution of effort etc. is never 100% consistent in a real-world project. Long-running branches can impact measurements when the metrics are calculated only on the main branch—the result from effort that has been invested earlier will not become visible for code-related metrics until a merge is performed. Also developer fluctuation and different characteristics of the components being built during the course of the project provoke changes where we assume homogeneous conditions. This may influence the results.

Differences between projects are surely multi-causal and the possible explanations given can only be assumptions of the most probable causes. Characteristics of projects and project phases were collected and written down ex post and this process may have been biased by the already known metrics results.

No comparison with functional size measurements, business value or estimates such as story points could be made for lack of such data. It is however clear that code-based metrics can only measure code and may, in edge-cases like an extremely verbose or inadequate implementation of a trivial functionality, yield results that are unsatisfying from a business perspective.

6 Conclusion

In this paper we have analyzed productivity values based upon different output metrics, in particular code change, and monthly effort data from four real-world software industry projects performed by the company itestra. We have shown that effort and output correlate within a given project and that significant and explainable productivity differences between projects and project phases exist. For instance, the transition from development to maintenance/bugfixing was in two projects associated with a drop in code output while the commit frequency remained stable. We conclude from our data that code change metrics can be (and we believe shound be) used for analyzing a project's productivity, if applied carefully and together with additional KPI. In contrast to story points they offer the possibility to benchmark against other projects. This is important as development teams have sometimes worked for years on the same system and comparison with benchmark values "outside the bubble" provides them with an independent view on possible productivity improvement.

In 15 years of our consulting work we have observed productivity values ranging from 9 *RFCC per person day* (in average over a team and year) at the bottom to 140 at the top for systems of similar size and complexity. Many researchers report a similarly large spread, e.g. in [6]. This cannot only be an effect of varying coding style or effort tracking, but we rather suspect differences in developer qualification and motivation, process overhead or quality deficits in the software. We consider the measurement and subsequent improvement of development productivity an enormous economic potential for the industry.

Future work should gather data on more projects and systematically track events throughout the course of the projects such as larger changes in team structure or development process.

7 Data availability

Due to confidentiality we cannot disclose the source code or additional information on the analyzed systems. The measurement data per time period and project used can be obtained at https://zenodo.org/doi/10.5281/zenodo.10265084.

References

1. International Software Benchmarking Standards Group, http://isbsg.org
2. Armour, P.G.: Beware of counting LOC. Communications of the ACM **47** (2004)

3. Barb, A., Neill, C., Sangwan, R., Piovoso, M.: A statistical study of the relevance of lines of code measures in software projects. Innovations in Systems and Software Engineering **10**(4) (2014). https://doi.org/10.1007/s11334-014-0231-5
4. Beck, K.: Outcome over output: Also impact and effort (2000), https://medium.com/@kentbeck_7670/outcome-over-output-also-impact-and-effort-8f9eb0ce0dbb
5. Beck, K., Fowler, M.: Planning Extreme Programming. Addison-Wesley (2000)
6. Boehm, B.: Software Cost estimation with COCOMO II (2002)
7. Deissenboeck, F., Juergens, E., Hummel, B., Wagner, S., Parareda, B.M.y., Pizka, M.: Tool support for continuous quality control. IEEE Software **25**(5) (2008). https://doi.org/10.1109/MS.2008.129
8. Drouin, N., Badri, M., Touré, F.: Analyzing software quality evolution using metrics: An empirical study on open source software. J. Softw. **8**(10) (2013)
9. Fernandez-Ramil, J., Lehman, M.: Metrics of software evolution as effort predictors - a case study (2000). https://doi.org/10.1109/ICSM.2000.883036
10. Forsgren, N., Storey, M.A., Maddila, C., Zimmermann, T., Houck, B., Butler, J.: The SPACE of developer productivity. ACM Queue **19** (2021)
11. Hill, P. (ed.): Practical Software Project Estimation: A Toolkit for Estimating Software Development Effort & Duration. McGraw Hill (2010)
12. Jones, C.: Estimating Software Costs : Bringing Realism to Estimating. McGraw-Hill,, New York:, 2nd ed. edn. (2007)
13. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: 2009 IEEE 31st International Conference on Software Engineering (2009). https://doi.org/10.1109/ICSE.2009.5070547
14. Kocaguneli, E., Misirli, A.T., Caglayan, B., Bener, A.: Experiences on developer participation and effort estimation. In: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (2011). https://doi.org/10.1109/SEAA.2011.71
15. Myers, E.: An o(nd) difference algorithm and its variations (1985)
16. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. (2005). https://doi.org/10.1109/ICSE.2005.1553571
17. Oliveira, E., Viana, D., Cristo, M., Conte, T.: How have software engineering researchers been measuring software productivity? - a systematic mapping study. In: Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2: ICEIS,. INSTICC, SciTePress (2017). https://doi.org/10.5220/0006314400760087
18. Mas y Parareda, B., Pizka, M.: Measuring productivity using the infamous lines of code metric. In: The First International Workshop on Software Productivity Analysis and Cost Estimation, Nagoya, Japan, Proceedings (2007)
19. Pizka, M., Panas, T.: Establishing economic effectiveness through software health-management. In: 1st International Workshop on Software Health Management (SHM 2009). Pasadena, CA (2009)
20. Richardson, L.F.: Fractals, vol. 1. Cambridge University Press (1993)
21. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering **37**(6) (2011). https://doi.org/10.1109/TSE.2010.81
22. Tan, T., Li, Q., Boehm, B., Yang, Y., He, M., Moazeni, R.: Productivity trends in incremental and iterative software development (2009). https://doi.org/10.1145/1671248.1671250

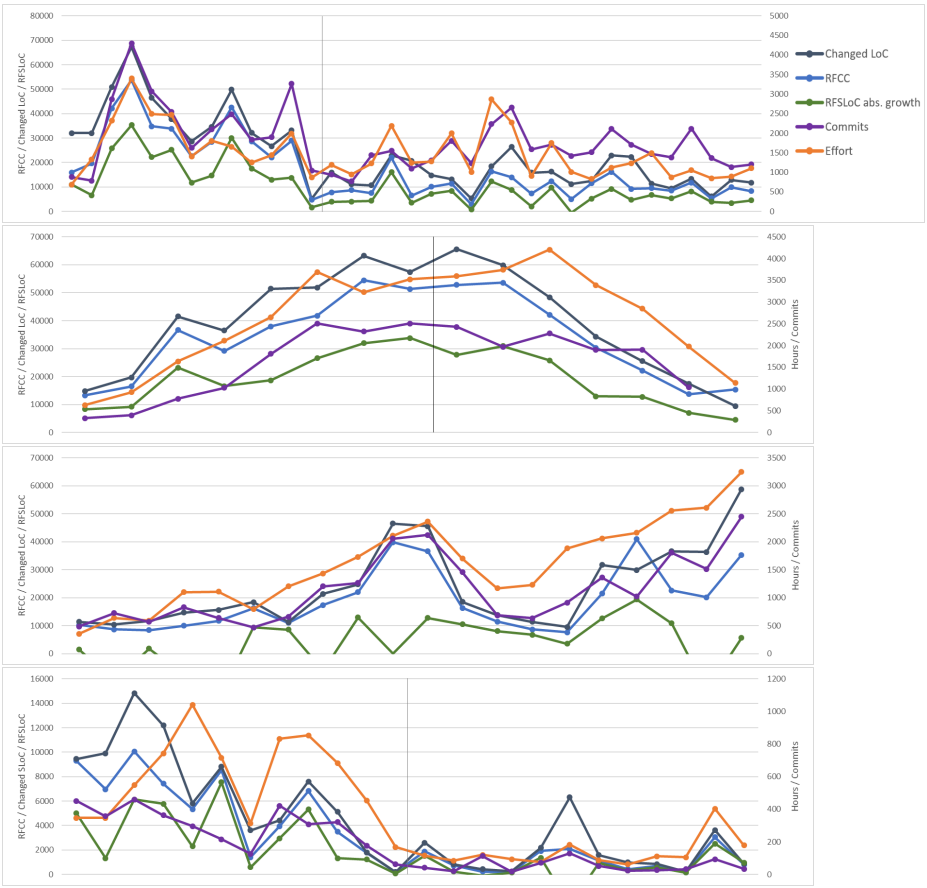

Fig. 1. Metric results for projects A to D.
Effort is in hours instead of days in this diagram in order to use a common scale with *Commits*. The y-axis starts at zero even if absolute growth happens to be negative. The vertical line marks the separation of project phases in A, B and D.

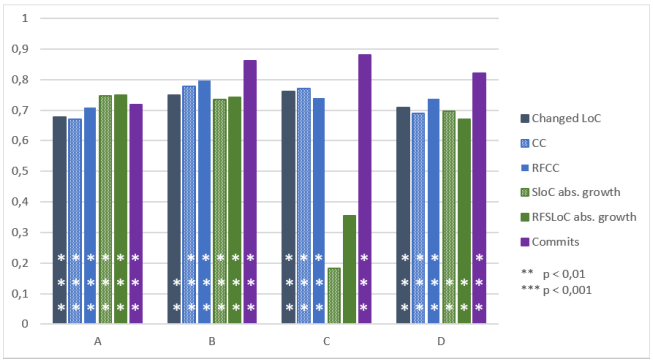

Fig. 2. Spearman correlation coefficients for different output metrics and projects correlated with effort

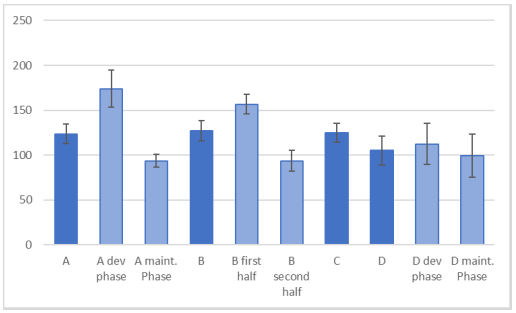

Fig. 3. Mean *Changed LoC per person day* for all projects including standard error

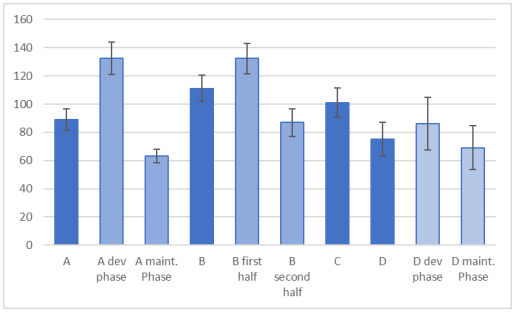

Fig. 4. Mean *RFCC per person day* for all projects including standard error

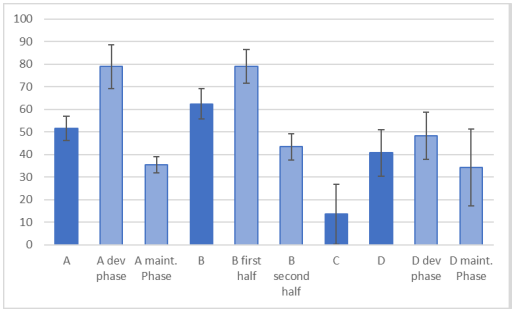

Fig. 5. Mean *RFSLoC absolute growth per person day* for all projects including standard error

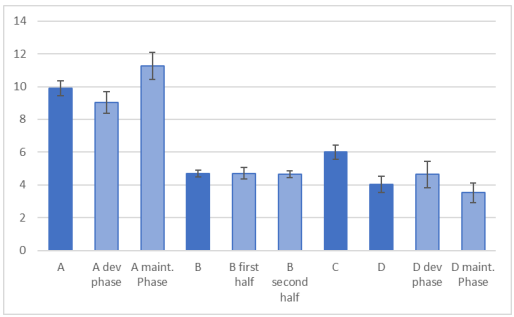

Fig. 6. Mean *Commits per person day* for all projects including standard error

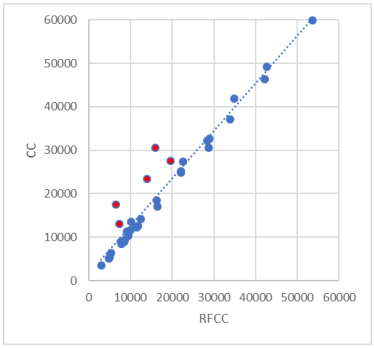

Fig. 7. *RFCC* vs *CC* values for project A, outliers marked in red