Software Productivity in DevOps

Qin Liu, Yidan Qin, Hongming Zhu*, Hongfei Fan
Tongji University, 4800 Cao'an Highway, Shanghai, China.

* Corresponding author. Tel.: +8613917590652; email: zhu_hongming@tongji.edu.cn
Manuscript submitted November 30, 2018; accepted December 5, 2018.
doi: 10.17706/jsw.14.3.129-137

Abstract: The investigation of multi-source, heterogeneous, multi-cycle data in DevOps has been attracting lots of attention in recent years. Although productivity is crucial for assuring instant release of DevOps, it has not been well studied based on merging effort features and unobserved cost features for open source software. An innovative software productivity estimation model in DevOps is proposed in this paper by recasting the definition of effort and cost. The proposed productivity model takes account of committed "outcomes" as cost instead of traditional man-month, and extended effort to consist of various commits (code, issue, scripts). Four open source projects are studied, with 95481 commits and 95828 issues in total. The experiment results illustrate the productivity changes with life cycle. The non-traditional code work ratio in productivity can represent iteration frequency of a software production and increases drastically before important releases. Thus we can monitor the life cycle and predicting large change of a production with productivity.

Key words: DevOps; open source code analysis; Software productivity; software project management.

1. Introduction

With software development entering the era of DevOps [1], [2], the investigation of multi-source, heterogeneous, multi-cycle data in DevOps has been attracting lots of attention in recent years [2], [3].

Productivity is crucial, since applications developed by the DevOps model need to be quickly released and promptly gathering user responses as the requirements basis for the next cycle.

Although there are some research published in schedule monitoring and resource management in DevOps model in industry and academia [4]-[6], however the software productivity has not been addressed particularly. The classic software productivity measurement and modelling models consist of product features and process features, and lack of consideration of related project resource features, such as interaction among stakeholders (e.g., feedback or behavioral data of professional users like developers and testers, and normal users). The multi-source, heterogeneous, multi-cycle data in DevOps provides a great potential to recast the definition of software productivity.

A software productivity model in DevOps is proposed in this paper, which can be considered as a confidence index to guide the resource management in the whole lifecycle. The proposed software productivity model recasts the effort estimation measurement and metrics by identifying a new set of effort features. It covers the effort of reuse of the source code, configuration for the deployment, derivation of test scripts, crawling of feedback, analysis of user behaviors etc. Computations based on data extracted from GitHub has been derived and executed accordingly.

1.1. Data Accessibility

With the continuous growth of the open source community, the cloud-based infrastructure enables a strong interdependence among DevOps-based software products and their design, development, operation, and maintenance, which provides a powerful accessibility of various project, process and resource feature data sets. For example, in an enterprise's internal development environment or open source software community, developers often provide a wide range of data in project homepages, version management, automation tools, communication tools, defect tracking systems, mailing lists, project forums, Wikis, social tags, user reviews, and so on. These content provides researchers and developers with not only a large number of software product feature data, software product development and operation and other process characteristics, but also resource characteristics such as reusable source code and automated scripts.

The proposed model in this paper provides a systematically and effectively approach for estimation in productivity for newly developed projects (such as entrepreneurial projects) based on DevOps.

1.2. Definition and Classification of Feature Sets

Feature sets of productivity in DevOps can be defined as the following:

Software product features: including the types of the software project, the development language used, the open source license compliance, the supported operating system, the database, and other static information of project: requirement repository information, code repository information, test repository information, defect repository information, task repository information, various automated script repository information and various types of automated script library information and other project characteristics; project development and operating environment and configuration characteristics; information of project developers, operation and maintenance personnel and user data of stakeholders.

Project process features: including the process raw data of the software lifecycle such as requirement library information, test library information, defect library information, operation library information, various types of automated script logs, product usage log information, and social network feedback information.

Project resource features: Including software project source code and project version information that are commonly found in the open source community; most projects include mailing list which contains communication information, APIs and related documentation, development environment configuration information, Issue information, and Stack Overflow Q&A Information (the Q&A information usually in project level, but a bit difficult to match to a particular code); very few projects have requirement information, interaction logs, running logs. At the same time, the open source software community also has a lot of social network features including follow, fork, etc.; there are a lot of time information for submit and update, which we can see a clearly rising and decay period within the project lifecycle.

2. Related Work

The initial research on software cost and productivity estimation focuses on how to build an estimation model and improve the accuracy of the estimation by experimenting with different modeling techniques. The development of software project development cost and productivity estimation theory has gone through the following technical stages: process-driven estimation modeling techniques such as SLIM models; empirically driven estimation modeling techniques such as Checkpoint, PRICE-S, SEER; and algorithm-based estimation modeling techniques such as COCOMO 81, COCOMOII; Bayesian-based models; artificial intelligence-based estimation modeling techniques, such as Case Based Reasoning, Regression trees, Optimized Set Reduction, Neurofuzzy cost models , Artificial Neural Network, and object-oriented model techniques, such as ObjectMetrix et al [7].

Existing software productivity estimation methods are based on the developer's experience, capabilities, and proficiency in the use of software development tools [8], usually in the form of productivity tables. Hence, measuring the cost and effort of different software projects is a crucial task [9]. However, the definition and calculation of software productivity have changed a lot in the DevOps environment. Cois [3] proposed the influence of software communication and data driving on software productivity in DevOps environment in 2014, and Casale [10] proposed the challenges and existing methods of software productivity in the new environment. These new definitions of software productivity combine the impact of new software development models in the DevOps environment on traditional software productivity. However, no in-depth scientific experiments have been conducted to further explore the correlation analysis and calculation of the impact of these interactive data on productivity. Since there is no further analysis and prediction of key factors affecting productivity in a given cycle, it is difficult to propose effective interventions for productivity changes, such as phased resource allocation analysis reports and projected improvements.

3. Definition of Productivity

DevOps-based software productivity includes not only the definition of the amount of development code (or function point) and the corresponding cost (people month) in the traditional productivity = output/cost (Effort/Cost), but can be recast as the follows: Effort includes

(1) Countable deliverables such as source code; (2) Reused (open source) source code; (3) Various types of automated scripts; (4) Reported defects; (5) Fixed defects.

Costs includes, but does not limited to, the monthly costs of the following activities:

(1) Development environment setup and deployment; (2) Using a crawler script to obtain reusable resource feature data; (3) Obtain software product, development process and available resource data from the version control system for development; (4) Compile with compiled scripts; (5) Use the test script to automatically test the compiled version; (6) Use the provision script to set up the operating environment; (7) Use deployment scripts for middleware and application deployment; (8) Use reptile scripts for application feedback acquisition; (9) Use analysis scripts for automatic feedback and program log analysis.

4. Experiments of Productivity

4.1. Data Sets and Experiments Environment

We use crawler script in Java 1.8 to achieve data in GitHub with RESTful API provided by GitHub.

In this paper, we use the project Eclipse Che as an example. Eclipse Che is an open source cloud IDE project, with about 300 watches, 4800 starts and 853 forks. We choose this project because it has typical features of a mature teamwork project, including code, dependency management system, version control system, test script, deploy script, developing environment document etc. 166 weeks' data are collected from this project, including issues, commits and tags.

We also include three other projects to the universality of our work, Microsoft Vscode, Atom and Adobe Brackets. For each commit, we focus on names of committed files, author, commit time and commit content. And for each issue, we focus on labels and contents. Amounts of commits, issues and tags used in this paper are listed in Table 1.

Table 1. Number of Commits, Issues and Tags for Each Project

Project	Commits	Issues	Tags
Eclipse Che	6246	9983	114
Vscode	36,531	53,931	101

Atom	35,009	17,476	474
Brackets	17,695	14,438	110

4.2. Computation of the Software Productivity in DevOps

After achieving the Git data, we map the data to the different types of work defined in Section 3.

1) Development environment setup and deployment. We use the number of lines included by "plugin" label for pom.xml, the core file of Maven, to stand for this part. Maven is used to managing different type files and plugins that can compile the certain type of files. It has the same effect as installing a new developing environment for certain language. So we use number of lines in pox.xml to represent this part.

2) Using a crawler script to obtain reusable resource feature data. It is not easy to get this kind of script. We now consider using whether the current project is forked from other projects to represent this kind of work.

3) Source code. This part including the number of lines in normal code file such as java, js, go, html, jsp, ts, sql, and css. Other kinds of source file can also be taken into consideration in other projects. This part is also used when calculate traditional work quantity, but now it's only a part of our work.

4) Compile with compiled scripts. Number of lines in pom.xml, which contain "dependency", "package" or "build" labels since these labels are used to descript how to build and package with the whole project.

5) Use the test script to automatically test the compiled version. We use number of lines in java files which are in test package.

6) Use the provision script to set up the operating environment. Not all projects provide script to set up environment. In our example project, it put the script and the introduction into readme file, guiding new developers to set up environment step by step. Thus we use the number of lines in readme file to represent this part of work.

7) Use deployment scripts for middleware and application deployment. Not all projects use middleware. In the example project, number of code lines in Docker related folder, such as "/dockerfiles", is used to represent this part of work since this project use Docker as runtime container.

8) Use reptile scripts for application feedback acquisition. We suggest that every issue is analyzed by the script. In this case, when an issue is created, developer will use script to achieve it. Thus we use the number issues created at that period to represent this work.

9) Use analysis scripts for automatic feedback and program log analysis. A closed issue must haves been analyzed so we use closed issues in a period to represent this work.

$$Productivity = \sum_j \text{normalize}(Effort_{ji}) \,/\, \sum_k \text{normalize}(Cost_{ki}) \tag{1}$$

$$\text{normalize}(Effort_{ji}) = Effort_{ji} \,/\, \max_{1 \le i \le weeks} Effort_{ji} \tag{2}$$

$$\text{normalize}(Cost_{ji}) = Cost_{ji} \,/\, \max_{1 \le i \le weeks} Cost_{ji} \tag{3}$$

Subscript *i* stands for week. Effort *j* is one of 5 kinds of effort above. And Cost *k* is one of 9 kinds of cost above.

However, it is not easy to get the data of cost. When we fetched data from open source community, it is easy to get the effort value since we can get number of lines of code changed from each commit. However, it is not easy to get the data of cost. A developer may not always work between his two commits and may take

leaves for several days. It is also possible that one committer commits for several developers because of authority problems. So we cannot find out how many human month one commit actually cost. Thus we only use data in effort part to verify the effectiveness of our definition. We use non-traditional work ratio to show that the variables in our definition really matters. We trade pure source code work as traditional work and define non-traditional work ratio as follow:

$$Non-TraditionalWorkRatio_i = \frac{\sum \text{normalize}(Work_{ji}) - \text{normalize}(SourceCodeWork_{ji})}{\text{normalize}(SourceCodeWork_{ji})} \quad (4)$$

4.3. Experiments Design and Execution

The propose of this experiment is verifying that the proportion of non-traditional code changes with the life cycle of the software project and this phenomenon is common in some projects.

First we fetched data (including issues, commits and tags) from git by API and washed the data. We divided the data according to the major versions of tags. In the major example project, it's 4.x.x, 5.x.x and 6.x.x. We count the weekly works and productivity. The costs and efforts were catalogued according to the definition in Section V. In Table2, we list the headers match with the definition in SectionⅢ.

Table 2. The Arrangement of Channels

Definition of productivity	Data column header
1) Development environment setup and deployment	developingEnvironmentEstablish
2) Using a crawler script to obtain reusable resource feature data	fetchReusableResourceWithCrawler
3) Source code	sourceCode
4) Compile with compiled scripts	autoCompile
5) Use the test script to automatically test the compiled version	autoTest
6) Use the provision script to set up the operating environment	runtimeEnviron-mentEstablish
7) Use deployment scripts for middleware and application deployment	deployMidwareAndApplication
8) Use reptile scripts for application feedback acquisition	getFeedback

Next we divided the data into Traditional Code Work and Non-traditional Code Work. Traditional Code Work consists of the code commits on the project files such as java, js, go, html, jsp, ts, sql, css, svg. Non-traditional Code Work consists of the cost defined in Section 5.

Then we worked out the proportion of (Non-traditional Code Work)/(Traditional Code Work) as the non-traditional work ratio defined in pervious section and show the result in linear chart.

Finally, we reflected the peak point in the linear chart to the time line of the example project to find if there were some significant events at those weeks. We also drew the linear chart of traditional code effort and productivity defined by us to make a compare.

To verify the universality of the change of non-traditional code ratio, we planned to repeat above steps on other three project with their own version number and file types respectively. For example, we can append IOS related files to Traditional Code Work in IOS related projects such as Atom.

5. Results Analysis

We first acted the above steps on major example, Eclipse Che and found some interesting character. Then, we repeated our experiment on Microsoft Vscode, Atom and Adobe Brackets.

5.1. Result in Eclipse Che

Fig. 1 is the lineage chart of the proportion of non-traditional code work and traditional code work. The Table 3 shows the relationship between the top point weeks and the released tags in these weeks. In major version 5.x.x, the first released version 5.0.0 comes following the highest point at week 16. Versions 5.1.x and 5.2.x also comes at peak week 19 and top week 21. The same phenomenon appears at week 25 for 5.4.x, week 27 for 5.5.0(5.5.0 is the only version for 5.5.x), week 34 for 5.10.0 and week 40 for 5.12.0 and 5.13.0.

We can assume that at the beginning of a major version, a large quantity of work must be put into getting feedback from users and reacting with them because of uncovered defects. Fixing defects frequently leads to the increase of non-traditional code work, which is shown as multi revision numbers in one minor version number and also the peak in the figure. After several minor versions, the project became stable and reliable. So developers did not donate much time to non-traditional code work when new minor versions were released. That's why the figure in later part doesn't change much.

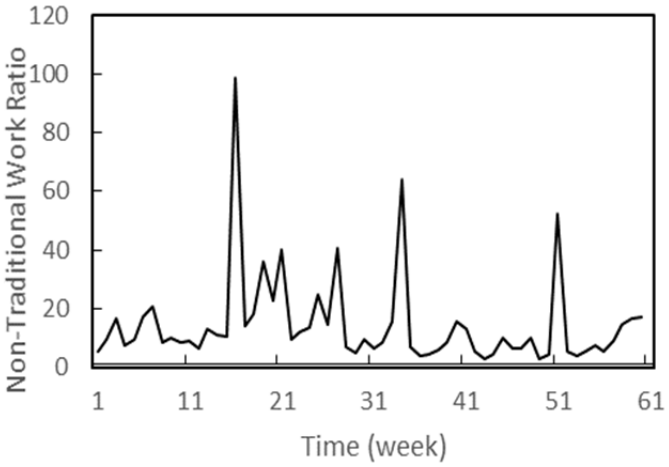


Fig. 1. Non-Traditional Work Ratio for Eclipse Che 5.x.x

month	day	developing Environment Establish	source Code	auto Compile	deployMid wareAndA pplication	get Feedback	analyzeF eedbackA ndLog
8	19	46	864	0	0	3	3
8	20	0	6	0	0	1	1
8	21	76	684	0	0	9	9
8	22	543	18471	975	5	9	9
8	23	1669	37620	1950	103	21	21
8	24	0	0	0	0	5	5
8	25	0	67	0	0	4	4
8	26	0	0	0	0	2	2
8	27	0	0	0	0	0	0
8	28	0	0	0	0	2	2
8	29	0	0	0	0	2	2
8	30	0	0	0	0	5	5
8	31	0	9	0	434	2	2
9	1	0	12	0	0	6	6
9	2	0	0	0	0	1	1
9	3	0	6	0	0	1	1
9	4	428	1349	0	1432	14	14
9	5	0	8	0	4	18	18
9	6	163	2241	0	203	13	13
9	7	155	3506	0	101	25	25
9	8	4	1439	0	59	15	15

Fig. 2. Part of Raw Data Leading Non-Traditional Work Ratio to change rapidly in vocation

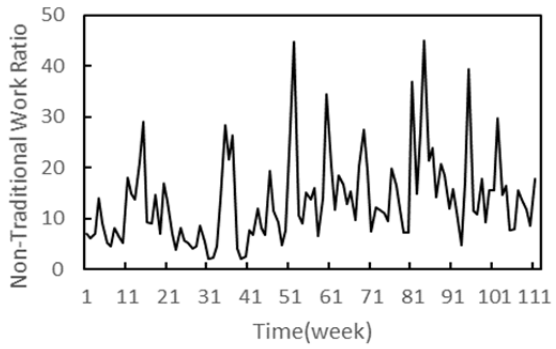


Fig. 3. Non-Traditional Work Ratio of Release 1.x.x of Microsoft Vscode

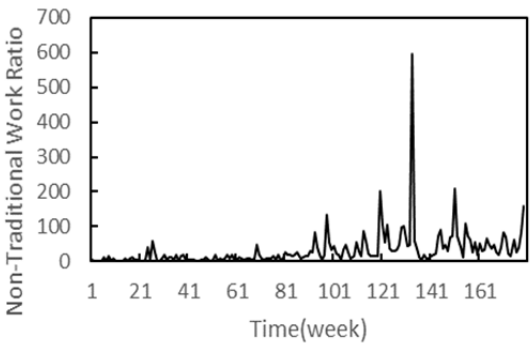


Fig. 4. Non-Traditional Work Ratio of Release 0.x.x of Atom

Table 3. Tags related with peak points

Week	Week Begin	Week End	Tag	Tag Created Time
16	20161228	20170103	5.0.0	2017-01-06
19	20170118	20170124	5.1.0	2017-01-18
			5.1.1	2017-01-20
			5.1.2	2017-01-24
21	20170201	20170207	5.2.0	2017-02-01
			5.2.1	2017-02-02
			5.2.2	2017-02-03

25	20170301	20170307	5.4.0 5.4.1	2017-03-02 2017-03-06
27	20170315	20170321	5.5.0	2017-03-15
34	20170503	20170509	5.10.0	2017-05-10
40	20170614	20170620	5.12.0 5.13.0	2017-06-14 2017-06-21
51	20170830	20170905	Vocation	

Here occurs an exception when it comes to week 51. That week started at Aug 30th and ended Sep 5rd. However, the previous version is released in Aug 23rd and the next version is released in Sep 20th. We checked our data again by day and found that there was almost no work at those days, as is shown in Fig. 2. So we suggest that the developers were on vacation after release on Aug 23rd. There might be some work left in week 50 so the ratio did not change much in week 50. In conclusion, we can distinguish this condition with important releases by absolute quantity of work.

5.2. Result on Three Other Projects

5.2.1. Result on microsoft vscode

In release 0.x.x, the ratio has limited weeks and does not change very significantly. However, when it comes to release 1.x.x, the ratio become volatile, as is shown in Fig. 3. The project has releases almost every month, so the ratio changes frequently and the figure differs a lot from other projects with long release interval. But the figure can still show some important release such as 1.8.0 in week 36, 1.11.0 in week 52 and 1.13.0 in week 60.

5.2.2. Result on atom

In release 0.x.x, the ratio remains steady at first and act as example project at the end about 100 weeks after the project started, as is shown in Fig. 4. In release 1.x.x, the peak points in Fig. 5 appear at the end of beta versions in each subversion release. Peak point of Week 46 is before release v1.8.0-beta4, followed by release v1.8.0; peak point of Week 72 is at release v1.12.0-beta7, followed by release v1.12.0 and peak point of Week 109 is at release v1.12.0-beta5, followed by beta6, beta7 and finally v1.12.0.

5.2.3. Result on adobe brackets

Non-Traditional Work Ratio of Adobe Brackets before release 1.x.x has few change except one peak of probable relation with Christmas holiday. In release 1.x.x, the ratio floats similarly with the example project. Fig. 6 succeeds in marking the release-1.2 on 19th Feb 2015 around Week 18, release-1.7 on 8th Jun 2016 around Week 87, release-1.9 around on 21st Mar 2017 around Week 126 and etc.

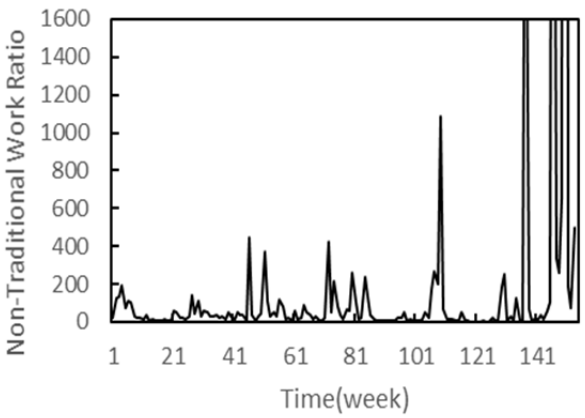


Fig. 5. Non-Traditional Work Ratio of Release 1.x.x of Atom

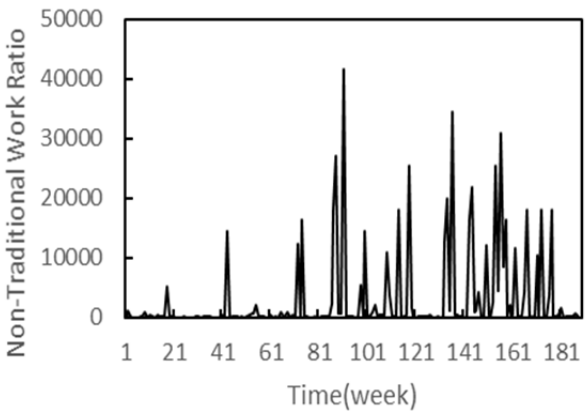


Fig. 6. Non-Traditional Work Ratio of Release 1.x.x of Adobe Brackets

6. Conclusion and Future Work

The line chart of non-traditional code ratio can show the character of lifecycle in a project. The ratio varies with life cycle. For most projects, the ratio remains a low level in developing period and rises obviously around important releases or holidays. For some projects with short interval between releases, the ratio changes quickly between low level and high level, so it is not easy to recognize the developing period but we can still find peak point around release date.

In future, we plan to cooperate with companies and get detail data of cost part to work out the real productivity. We can also get the lifecycle schedule to work detail relation between lifecycle and the productivity. Meanwhile, more research is needed on how we can use the ratio or the productivity to improve the developing process. We may simulate investing more resource when the ratio or the productivity changes to work out whether it is possible to reduce the software cost or improve the software quality.

Acknowledgment

This work was supported in part by the Ministry of Science and Technology of China under Grant No. 2016YFB1000805, the National Natural Science Foundation of China under Grant No. 61702374, the Shanghai Sailing Program under Grant No. 17YF1420500 and the Fundamental Research Funds for the Central Universities.

References

[1] Wettinger, J., Andrikopoulos, V., & F. Leymann, Automated capturing and systematic usage of devops knowledge for cloud applications. *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, Tempe (pp. 60-65).

[2] Wettinger, J., Breitenbücher, U., & Leymann, F. (2014). devopslang – Bridging the gap between development and operations. *Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC)* (pp. 108-122).

[3] Cois, C. A., Yankel, J., & Connell, A. (2014). Modern DevOps: Optimizing software development through effective system interactions. *Proceedings of the 2014 IEEE International Professional Communication Conference (IPCC)* (pp. 1-7).

[4] Miglierina, M., & Tamburri, D. A. (2017). Towards omnia: A monitoring factory for quality-aware devops. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (pp. 145-150).

[5] Peuster, M., & Karl, H. (2017). Profile your chains, not functions: Automated network service profiling in DevOps environments. *Proceedings of the 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks* (pp. 1-6).

[6] Guerriero, M., Ciavotta, M., Gibilisco, G. P., & Ardagna, D. (2015). A model-driven devops framework for QoS-aware cloud applications. *Proceedings of the 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (pp. 345-351).

[7] Ani, Z. C., Basri, S., & Sarlan, A. (2017). A reusability assessment of UCP-based effort estimation framework using object-oriented approach. *Journal of Telecommunication, Electronic and Computer Engineering*, *9*, 111-114.

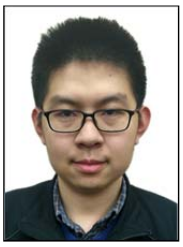
[8] Vasantrao, K. V. (2011). Enhance accuracy in Software cost and schedule estimation by using uncertainty analysis and assessment' in the system modeling process. *International Journal of Research and Innovation in Computer Engineering*.

[9] Bilgaiyan, S., Mishra, S., & Das, M. (2016). A review of software cost estimation in agile software development using soft computing techniques. *Proceedings of the 2016 2nd International Conference on Computational Intelligence and Networks (CINE)* (pp. 112-117).

[10] Casale, G., Chesta, C., Deussen, P., Nitto, E. D., Gouvas, P., Koussouris, S., Stankovski, V., Symeonidis, A., Vlassiou, V., Zafeiropoulos, A., & Zhao, Z. (2016). Current and future challenges of software engineering for services and applications. *Procedia Computer Science*, *97*, 34-42.

Qin Liu was born in Xingjiang, China, in 1976. She received B.E. degree in electronic Engineering from Dalian University of Technology, China, in 1999, and the Ph.D. degree from Northumbria University, UK, in 2006.

Since 2007, she has been a professor with the School of Software Engineering, Tongji University, China. Her research interests include Software cost estimation and software engineering.



Yidan Qin, born in April 23, 1993, Shenyang, China, got bachelor degree of engineering in 2016 in School of Software Engineering, Tongji University, Shanghai, China.
He is now working on the *Software Intelligent Development Method and Environment Based on Big Data*.

Mr. Qin is now studying for master degree under the guidance of Professor Liu in Tongji.

Hongming Zhu was born in Baotou, China, in 1980. He received the B.E. degree in software engineering from Tongji University, China, in 2002, and the Ph.D. degree from Bolton University, UK, in 2017.

Since 2015, he has been an associate professor with the School of Software Engineering, Tongji University, China. His research interests include distributed graph analysis and data analysis.

Hongfei Fan was born in Suzhou, China, in 1985. He received the B.E. degree in software engineering from Tongji University, China, in 2007, and the Ph.D. degree in computer science from Nanyang Technological University, Singapore, in 2013.

Since 2014, he has been an assistant professor with the School of Software Engineering, Tongji University, China. His research interests include computer-supported cooperative work (CSCW) and software engineering.

Dr. Fan is a member of ACM, IEEE and CCF. He received the Best Paper Awards from IEEE CSCWD 2012 and ACM SAC 2012.