Beyond Code

Towards Intelligent Collaboration Tools

Vladimir Kovalenko

BENEVOL 2021

08.12.2021

RESEARCH

Beyond Code

Towards Intelligent Collaboration Tools

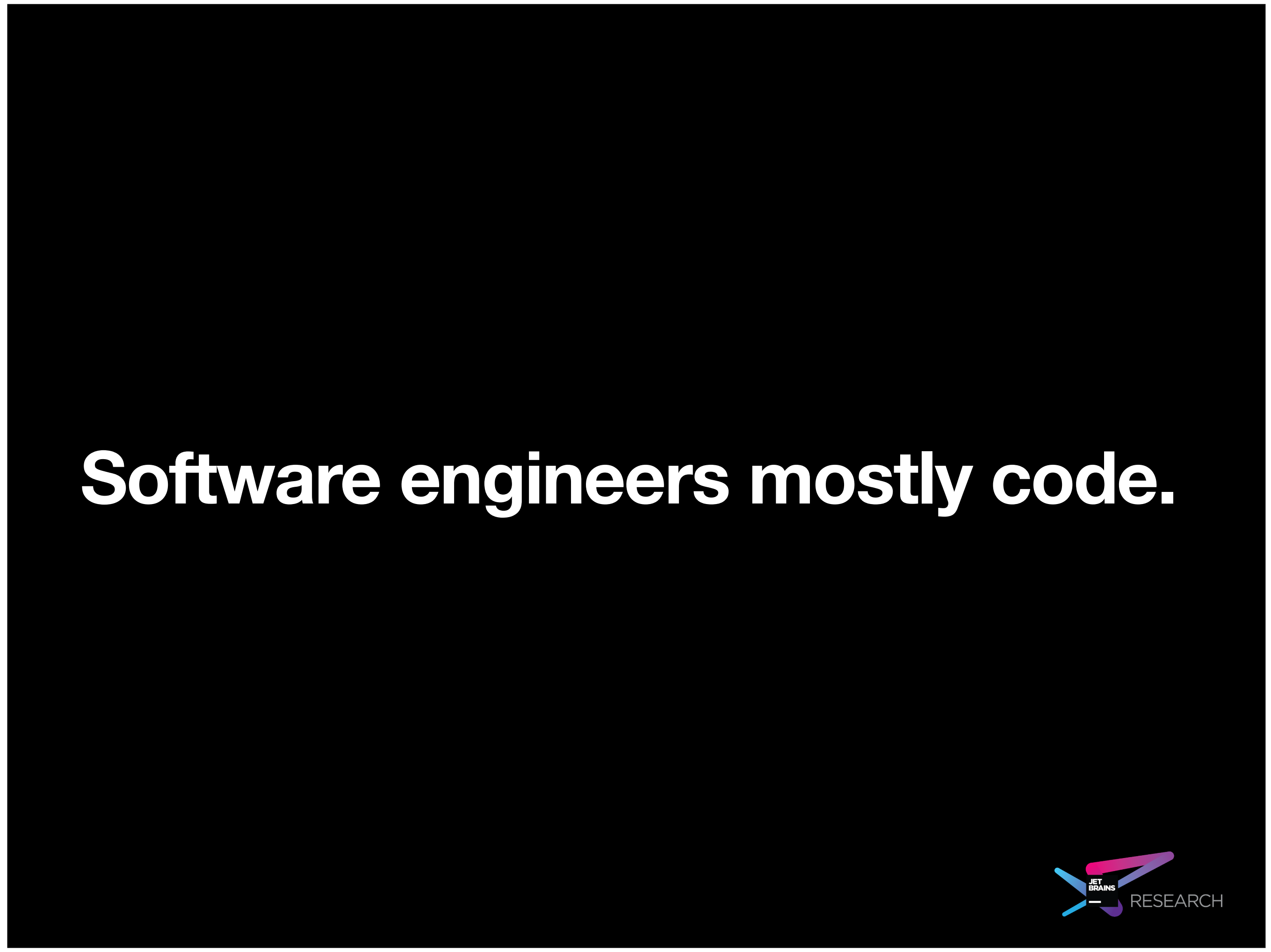
Vladimir Kovalenko

BENEVOL 2021

08.12.2021

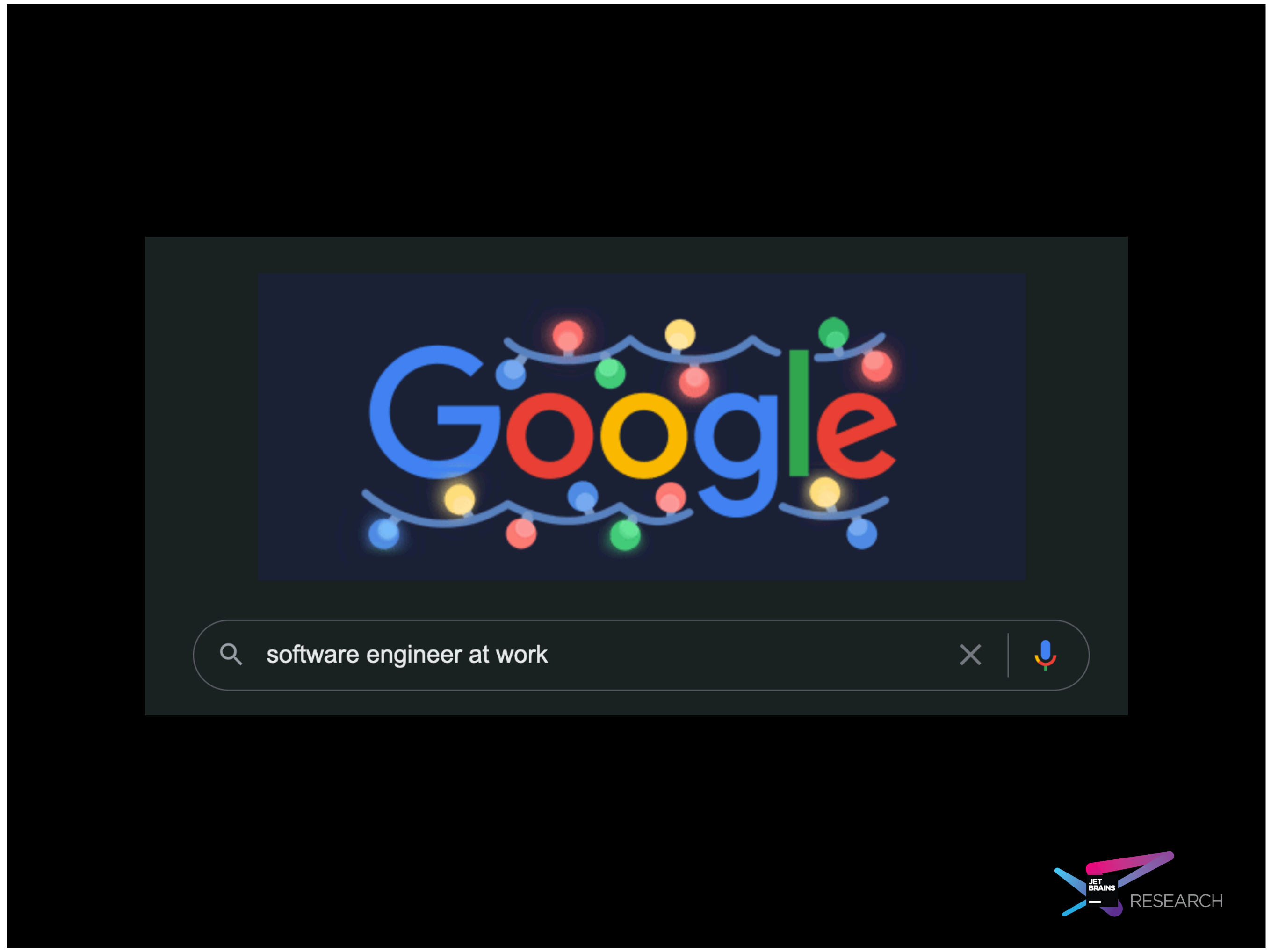
RESEARCH

Introduce myself

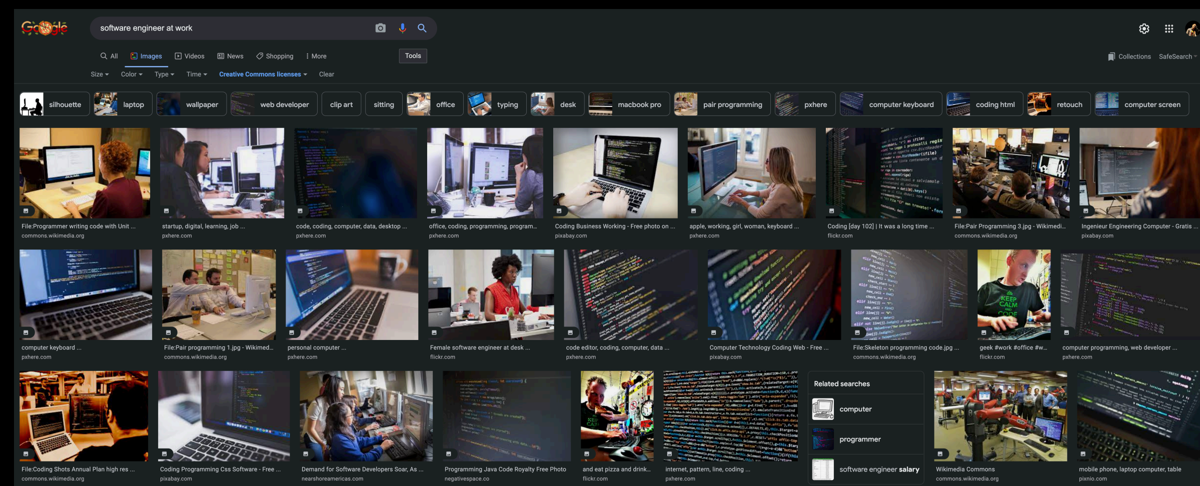


Let me start with a statement:
Software engineers mostly code.

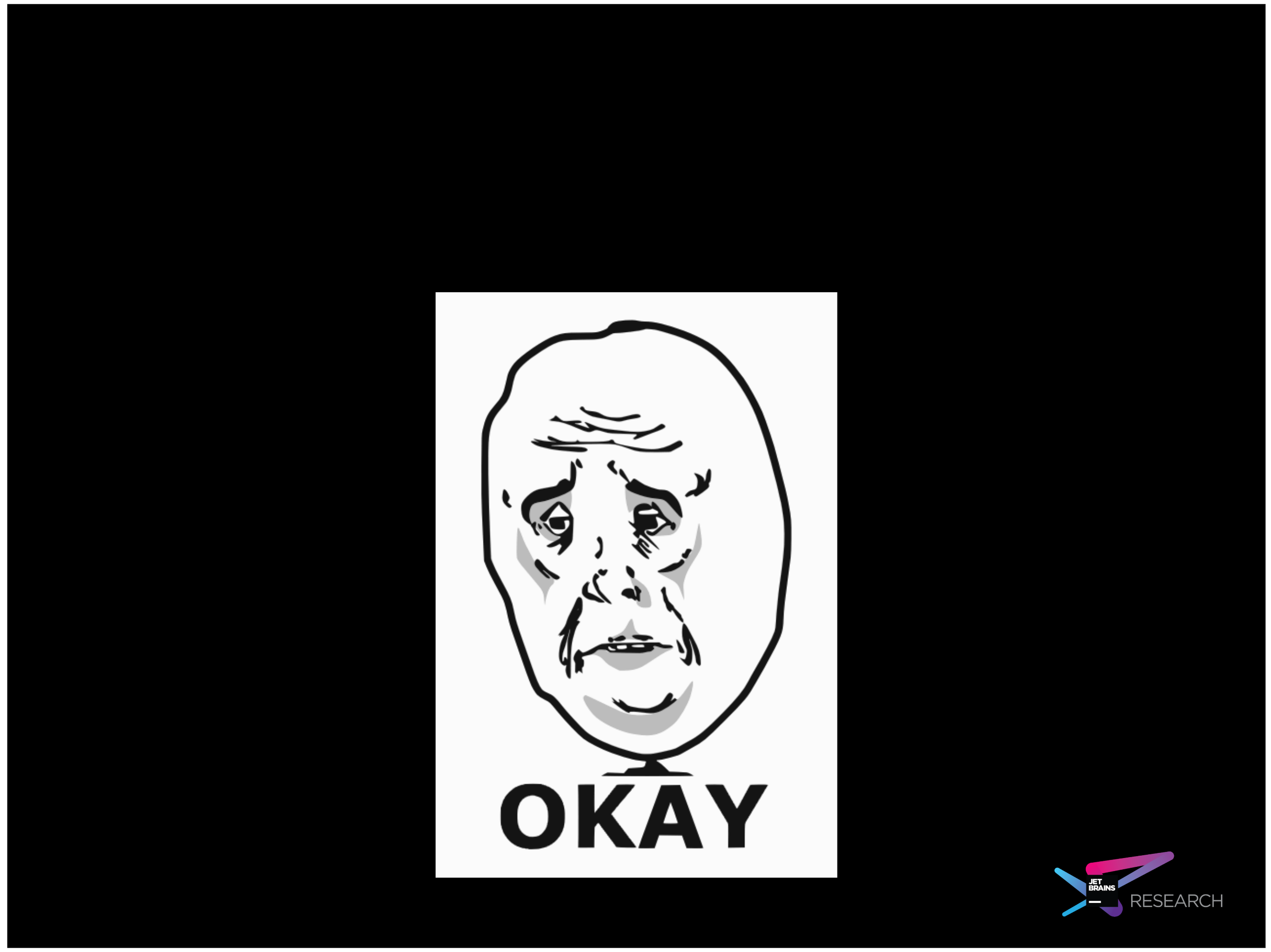
Do you agree? I don't. Let me try to disprove this statement.



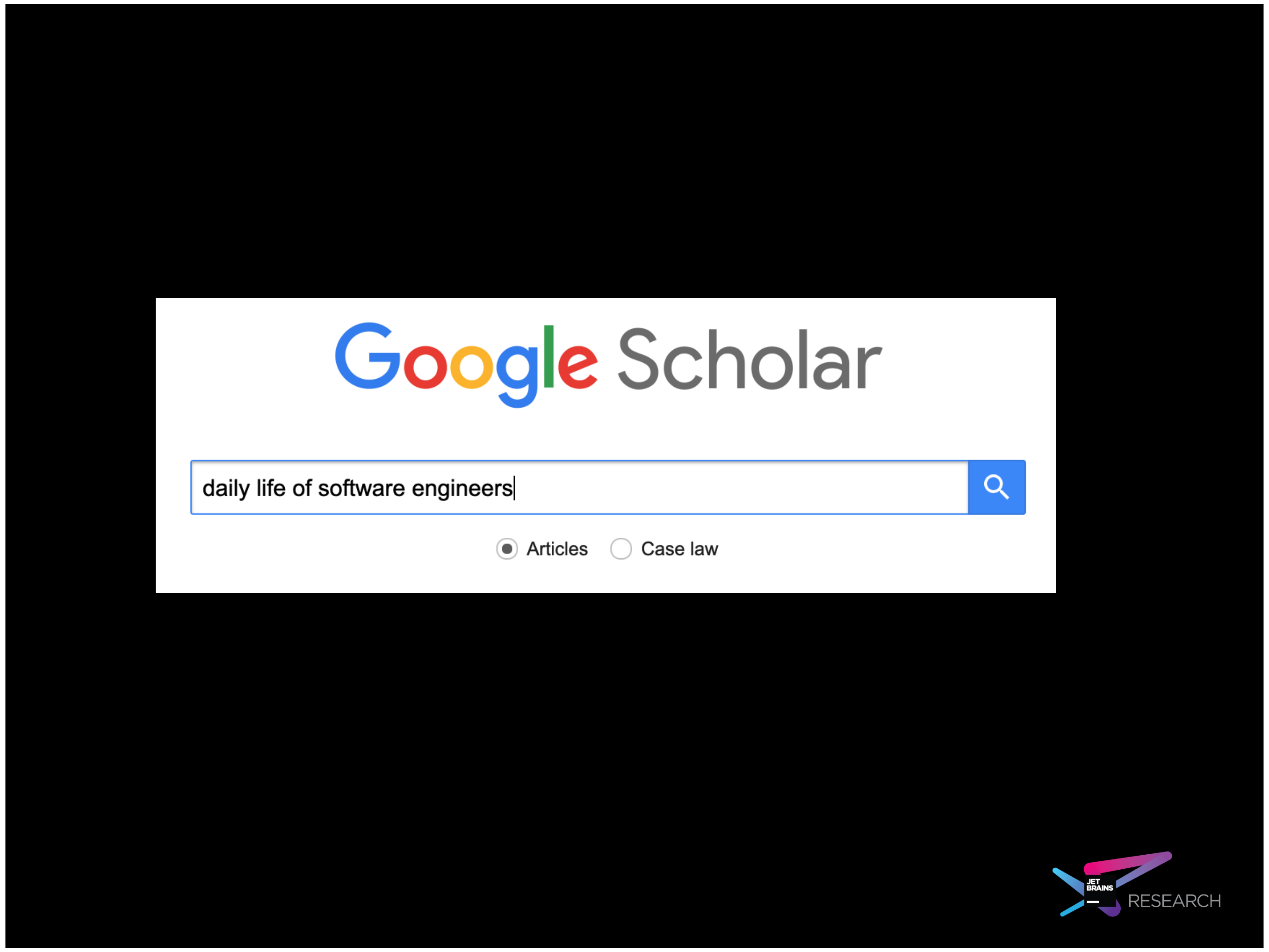
Let’s ask Google.



If we look up “software engineer at work”, we see lots of code and little beyond that in their screens.



Of course, one million web pages cannot be wrong. It’s not so easy to disprove this.
At this point, we could stop trying and call it a day, but we’re scientists…



…So let's try again.

We are going to find a few interesting articles, over at least three last decades, reporting on studies aimed at figuring out what the developers actually do at work.

Mean and relative time spent on activities on developers' previous workdays (WD). The left number in a cell indicates the average relative time spent (in percent) and the right number in a cell the absolute average time spent (in minutes).

Activity Category	All 100% (N=5928)		Typical WD 64% (N=3750)		Atypical WD 36% (N=2099)		Good WD 61% (N=3028)		Bad WD 39% (N=1970)	
	pct	min	pct	min	pct	min	pct	min	pct	min
Development-Heavy Activities										
Coding (reading or writing code and tests)	15%	84	17%	92	13%	70	18%	96	11%	66
Bugfixing (debugging or fixing bugs)	14%	74	14%	77	12%	68	14%	75	13%	72
Testing (running tests, performance/smoke testing)	8%	41	8%	44	7%	36	8%	43	7%	38
Specification (working on/with requirements)	4%	20	3%	17	4%	25	4%	20	4%	20
Reviewing code	5%	25	5%	26	4%	23	4%	24	5%	26
Documentation	2%	9	1%	8	2%	10	2%	9	2%	8
Collaboration-Heavy Activities										
Meetings (planned and unplanned)	15%	85	15%	82	17%	90	14%	79	18%	95
Email	10%	53	10%	54	10%	54	9%	52	10%	57
Interruptions (impromptu sync-up meetings)	4%	24	4%	25	4%	22	4%	22	5%	28
Helping (helping, managing or mentoring people)	5%	26	5%	27	5%	25	5%	26	5%	28
Networking (maintaining relationships)	2%	10	2%	9	2%	12	2%	11	2%	10
Other Activities										
Learning (honing skills, continuous learning, trainings)	3%	17	3%	14	4%	22	3%	19	3%	16
Administrative tasks	2%	12	2%	11	3%	14	2%	11	3%	15
Breaks (bio break, lunch break)	8%	44	8%	44	8%	45	8%	44	8%	45
Various (*e.g.* traveling, planning, infrastructure set-up)	3%	21	3%	17	5%	27	3%	19	4%	25
Total	**9.08** hours		**9.12** hours		**9.05** hours		**9.17** hours		**9.15** hours	

Meyer, Andre, et al. "Today was a good day: The daily life of software developers." IEEE Transactions on Software Engineering (2019, preprint).

RESEARCH

Let's take a quick look at some of the results from these papers. This table from Andre Meyer's work presents the distribution of activities reported by almost 6K professional developers.

Mean and relative time spent on activities on developers' previous workdays (WD). The left number in a cell indicates the average relative time spent (in percent) and the right number in a cell the absolute average time spent (in minutes).

Activity Category	All 100% (N=5928)		Typical WD 64% (N=3750)		Atypical WD 36% (N=2099)		Good WD 61% (N=3028)		Bad WD 39% (N=1970)	
	pct	min	pct	min	pct	min	pct	min	pct	min
Development-Heavy Activities										
Coding (reading or writing code and tests)	15%	84	17%	92	13%	70	18%	96	11%	66
Bugfixing (debugging or fixing bugs)	14%	74	14%	77	12%	68	14%	75	13%	72
Testing (running tests, performance/smoke testing)	8%	41	8%	44	7%	36	8%	43	7%	38
Specification (working on/with requirements)	4%	20	3%	17	4%	25	4%	20	4%	20
Reviewing code	5%	25	5%	26	4%	23	4%	24	5%	26
Documentation	2%	9	1%	8	2%	10	2%	9	2%	8
Collaboration-Heavy Activities										
Meetings (planned and unplanned)	15%	85	15%	82	17%	90	14%	79	18%	95
Email	10%	53	10%	54	10%	54	9%	52	10%	57
Interruptions (impromptu sync-up meetings)	4%	24	4%	25	4%	22	4%	22	5%	28
Helping (helping, managing or mentoring people)	5%	26	5%	27	5%	25	5%	26	5%	28
Networking (maintaining relationships)	2%	10	2%	9	2%	12	2%	11	2%	10
Other Activities										
Learning (honing skills, continuous learning, trainings)	3%	17	3%	14	4%	22	3%	19	3%	16
Administrative tasks	2%	12	2%	11	3%	14	2%	11	3%	15
Breaks (bio break, lunch break)	8%	44	8%	44	8%	45	8%	44	8%	45
Various (*e.g.* traveling, planning, infrastructure set-up)	3%	21	3%	17	5%	27	3%	19	4%	25
Total	**9.08** hours		**9.12** hours		**9.05** hours		**9.17** hours		**9.15** hours	

Meyer, Andre, et al. "Today was a good day: The daily life of software developers." IEEE Transactions on Software Engineering (2019, preprint).

Less than half of time is spent on code-related activities.

RESEARCH

If we take a close look, we see that less that half of all time is dedicated to code-related activities.

A few other similar studies, that we will not dive deep into to save time, present a similar overall picture.

Mean and relative time spent on activities on developers' previous workdays (WD). The left number in a cell indicates the average relative time spent (in percent) and the right number in a cell the absolute average time spent (in minutes).

Activity Category	All 100% (N=5928)		Typical WD 64% (N=3750)		Atypical WD 36% (N=2099)		Good WD 61% (N=3028)		Bad WD 39% (N=1970)	
	pct	min	pct	min	pct	min	pct	min	pct	min
Development-Heavy Activities										
Coding (reading or writing code and tests)	15%	84	17%	92	13%	70	18%	96	11%	66
Bugfixing (debugging or fixing bugs)	14%	74	14%	77	12%	68	14%	75	13%	72
Testing (running tests, performance/smoke testing)	8%	41	8%	44	7%	36	8%	43	7%	38
Specification (working on/with requirements)	4%	20	3%	17	4%	25	4%	20	4%	20
Reviewing code	5%	25	5%	26	4%	23	4%	24	5%	26
Documentation	2%	9	1%	8	2%	10	2%	9	2%	8
Collaboration-Heavy Activities										
Meetings (planned and unplanned)	15%	85	15%	82	17%	90	14%	79	18%	95
Email	10%	53	10%	54	10%	54	9%	52	10%	57
Interruptions (impromptu sync-up meetings)	4%	24	4%	25	4%	22	4%	22	5%	28
Helping (helping, managing or mentoring people)	5%	26	5%	27	5%	25	5%	26	5%	28
Networking (maintaining relationships)	2%	10	2%	9	2%	12	2%	11	2%	10
Other Activities										
Learning (honing skills, continuous learning, trainings)	3%	17	3%	14	4%	22	3%	19	3%	16
Administrative tasks	2%	12	2%	11	3%	14	2%	11	3%	15
Breaks (bio break, lunch break)	8%	44	8%	44	8%	45	8%	44	8%	45
Various (*e.g.* traveling, planning, infrastructure set-up)	3%	21	3%	17	5%	27	3%	19	4%	25
Total	**9.08** hours		**9.12** hours		**9.05** hours		**9.17** hours		**9.15** hours	

RESEARCH

OK, but what else do developers do?

Mean and relative time spent on activities on developers' previous workdays (WD). The left number in a cell indicates the average relative time spent (in percent) and the right number in a cell the absolute average time spent (in minutes).

Activity Category	All 100% (N=5928)		Typical WD 64% (N=3750)		Atypical WD 36% (N=2099)		Good WD 61% (N=3028)		Bad WD 39% (N=1970)	
	pct	min	pct	min	pct	min	pct	min	pct	min
Development-Heavy Activities										
Coding (reading or writing code and tests)	15%	84	17%	92	13%	70	18%	96	11%	66
Bugfixing (debugging or fixing bugs)	14%	74	14%	77	12%	68	14%	75	13%	72
Testing (running tests, performance/smoke testing)	8%	41	8%	44	7%	36	8%	43	7%	38
Specification (working on/with requirements)	4%	20	3%	17	4%	25	4%	20	4%	20
Reviewing code	5%	25	5%	26	4%	23	4%	24	5%	26
Documentation	2%	9	1%	8	2%	10	2%	9	2%	8
Collaboration-Heavy Activities										
Meetings (planned and unplanned)	15%	85	15%	82	17%	90	14%	79	18%	95
Email	10%	53	10%	54	10%	54	9%	52	10%	57
Interruptions (impromptu sync-up meetings)	4%	24	4%	25	4%	22	4%	22	5%	28
Helping (helping, managing or mentoring people)	5%	26	5%	27	5%	25	5%	26	5%	28
Networking (maintaining relationships)	2%	10	2%	9	2%	12	2%	11	2%	10
Other Activities										
Learning (honing skills, continuous learning, trainings)	3%	17	3%	14	4%	22	3%	19	3%	16
Administrative tasks	2%	12	2%	11	3%	14	2%	11	3%	15
Breaks (bio break, lunch break)	8%	44	8%	44	8%	45	8%	44	8%	45
Various (*e.g.* traveling, planning, infrastructure set-up)	3%	21	3%	17	5%	27	3%	19	4%	25
Total	**9.08** hours		**9.12** hours		**9.05** hours		**9.17** hours		**9.15** hours	

RESEARCH

They mostly collaborate.

This includes communicating in meetings and emails and reviewing code. Some other things not explicitly reflected in this example include chatting to colleagues in messenger workspaces, managing the issue tracker, and other things like that.

Software engineers ~~mostly~~ code.

RESEARCH

Now let’s correct the statement a bit.

Software engineers ~~mostly~~ code.

Also, they exchange information.

And extend it.

In the modern world, all these collaborative activities are supported by dedicated tools, that are often tailored specifically to software engineering.

Collaboration tools

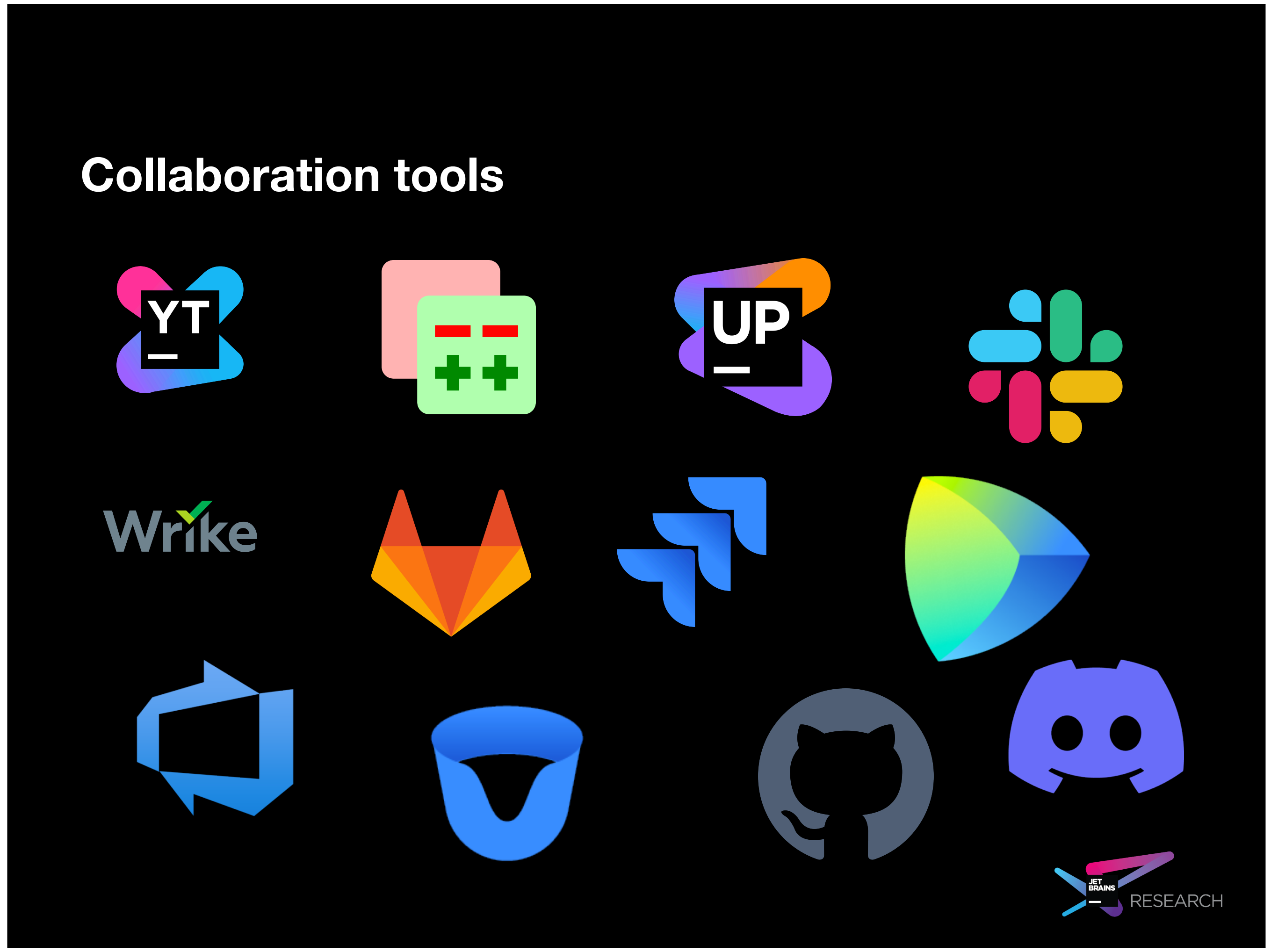
There are issue trackers and project management tools like YouTrack or Jira…

Collaboration tools

There are code review tools like Gerrit, Upsource, or Crucible…

Collaboration tools

There are messenger workspaces and meeting software like Slack, Discord, or Google Meet



Also, there are integrated solutions, incorporating the capabilities of several other tools or classes of tools, such as GitHub, JetBrains Space, or GitLab.

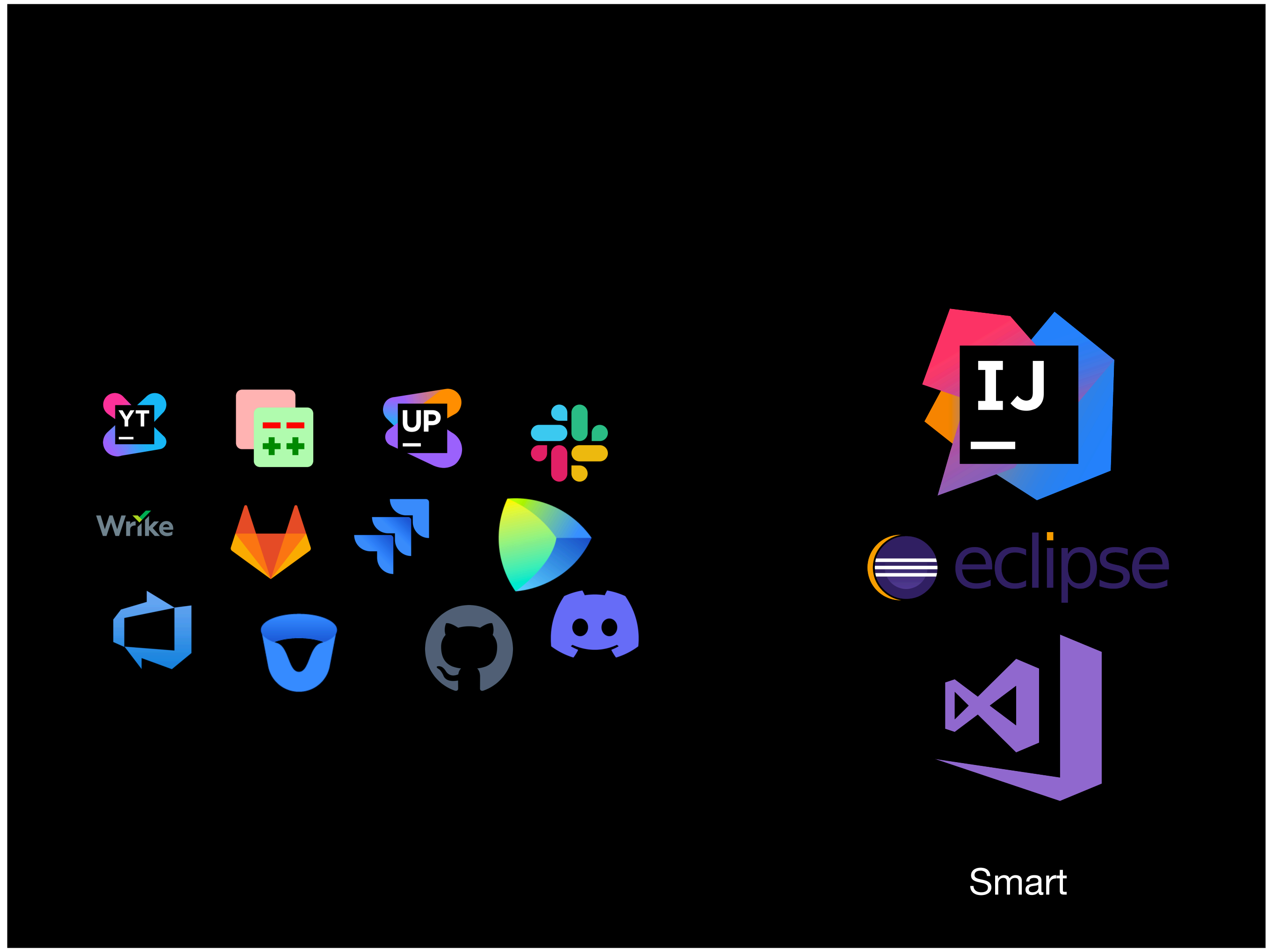
The list is of course not complete, but it should give a general idea: there are many dedicated collaboration tools in use today.

Collaboration tools

Let’s take a look at them from a different perspective: how do they compare with another class of mainstream developer tools?

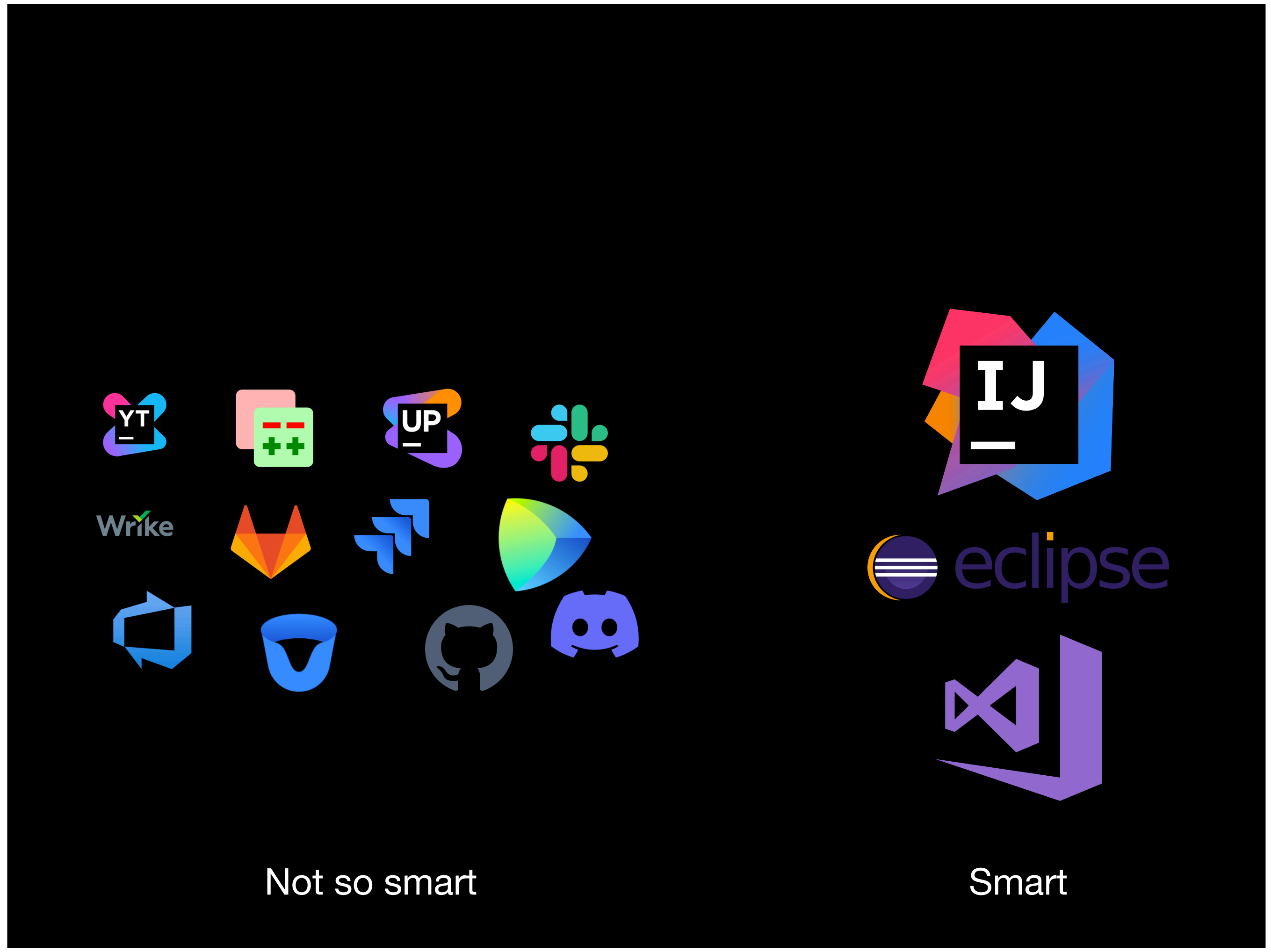
Collaboration tools vs IDEs

I mean the IDEs.



IDEs are really smart tools. They save thousands and thousands of human-hours of work every day by simplifying complex operations like refactorings, help ensure code quality through static analysis, and support developers in working with myriads of other tools and frameworks.

To support all these things, IDEs build and maintain complex representations of the code and its dependencies and efficiently operate with them. They are indeed some of the most complex pieces of software out there.

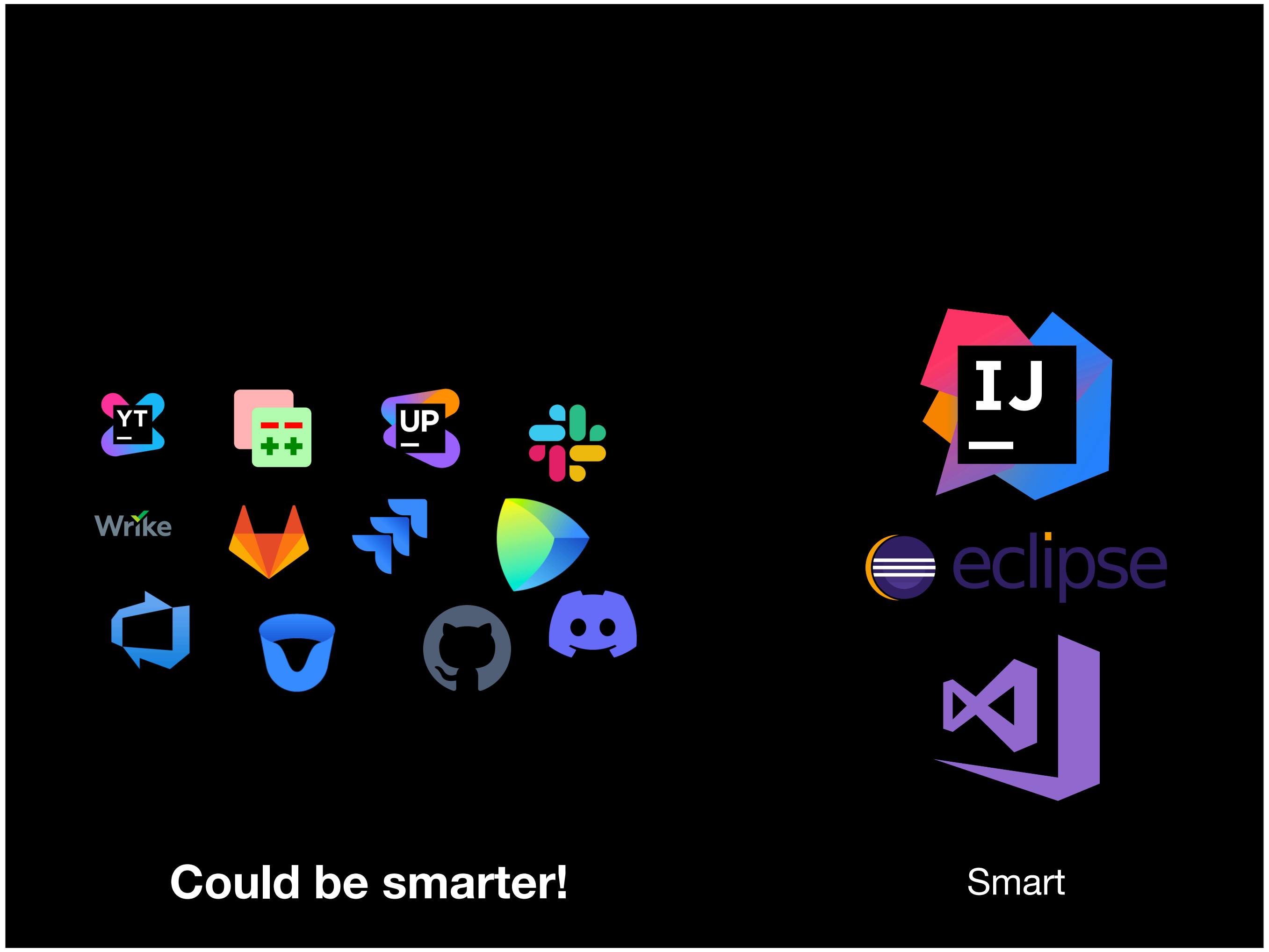


In contrast, the collaboration tools of today are not that smart.
To exaggerate a bit, at a very high level, they are not much different from bulletin board engines.

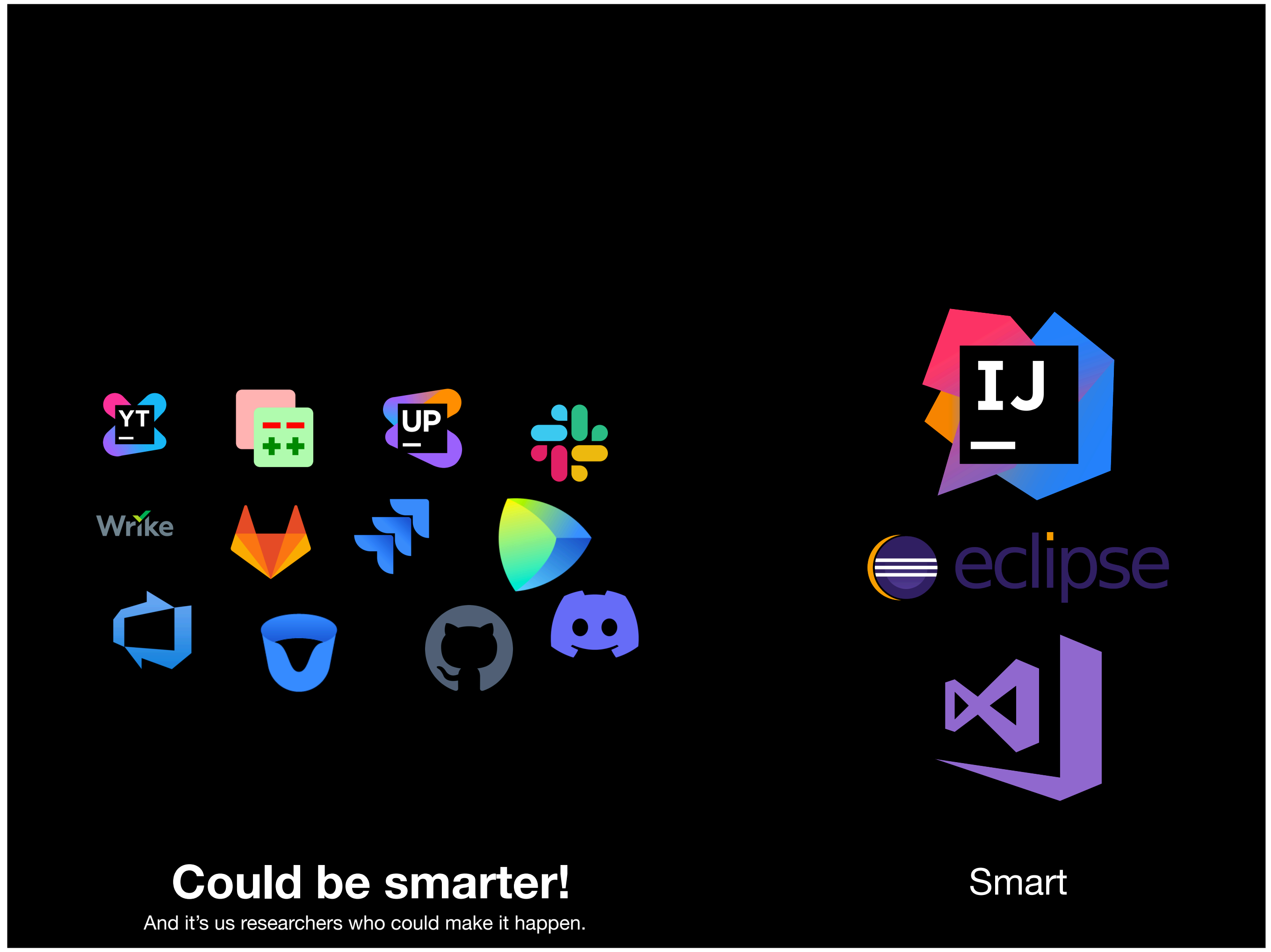
Of course, they are also beautiful and complex pieces of software, designed to be reliable, distributed, and fast, and they also do help their users a lot, and mostly have brilliantly designed user experience.

All of this brings me to this really important point:

However, while IDEs deeply analyse the medium they are designed to work with — I mostly mean code — the collaboration tools do not really deeply analyse the data they operate with.



Collaboration tools in software engineering could be smarter.

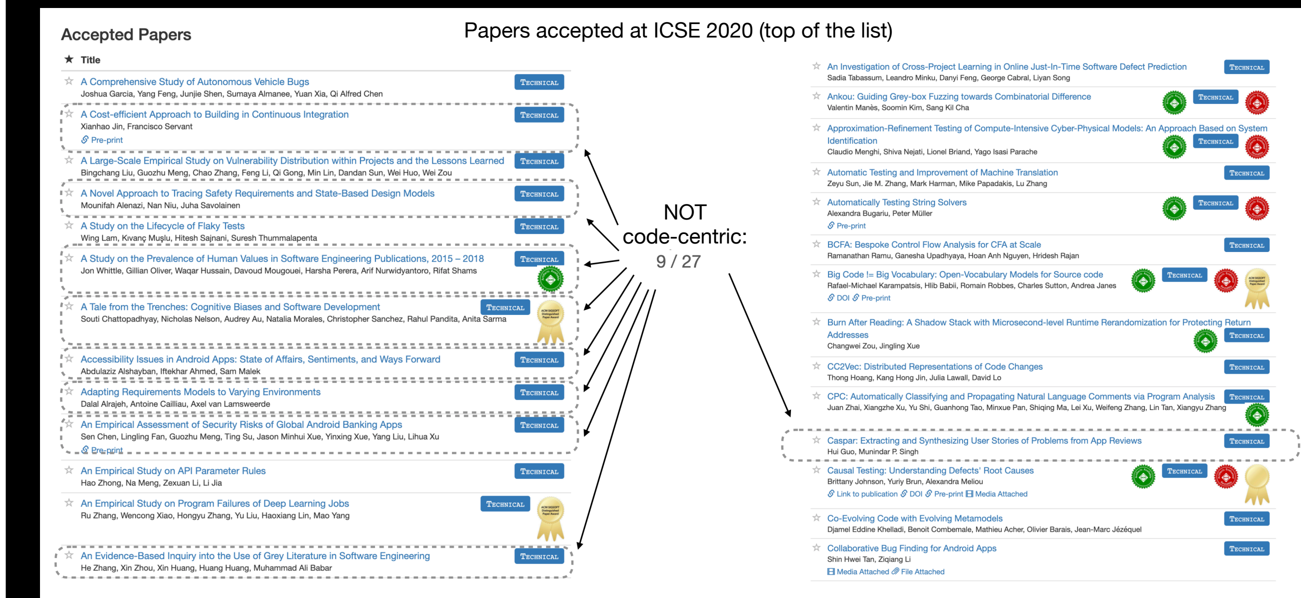


And it’s us researchers who could make it happen.

Collaboration tools could be smarter!

Let me try to amplify this message and try to convince you that we should really pay more attention to collaboration tools.

Argument 1: we are biased towards coding



Not so long ago, while thinking about this difference between IDEs and collaboration tools I've just talked about, I've tried to kind of prove myself wrong and collect some numbers.

I opened the then-recent list of papers accepted to ICSE, and tried to count the papers that are focused on code and those that are not. In the top of the list, two of three papers were about code in one way or another.

Argument 1: we are biased towards coding

64	Plug the Database & Play With Automatic Testing: Improving System Testing by Exploiting Persistent Data	
65	Predicting Code Context Models for Software Development Tasks	1
66	Prober: Practically Defending Overflows with Page Protection	
67	Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance	
68	Representing and Reasoning about Dynamic Code	1
69	Retrieve and Refine: Exemplar-based Neural Comment Generation	1
70	Revisiting the relationship between fault detection, test adequacy criteria, and test set size.	1
71	SADT: Syntax-Aware Differential Testing of Certificate Validation in SSL/TLS Implementions	
72	Scalable Multiple-View Analysis of Reactive Systems via Bidirectional Model Transformations	1
73	Scaling Client-Specific Equivalence Checking via Impact Boundary Search	
74	SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis	1
75	Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for AndroidExperience	1
76	Stay Professional and Efficient: Automatically Generate Titles for Your Bug Reports	
77	Subdomain-Based Generality-Aware Debloating	1
78	Summary-Based Symbolic Evaluation for Smart Contracts	1
79	Synthesis of Infinite-State Systems with Random Behavior	
80	Synthesis-Based Resolution of Feature Interactions in Cyber-Physical Systems	
81	Team Discussions and Dynamics During DevOps Tool Adoptions in OSS Projects	
82	Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study	
83	TestMC: Testing Model Counters using Differential and Metamorphic TestingExperience	1
84	The Impact of Generic Data Structures: Decoding the Role of Lists in the Linux Kernel	1
85	Towards Generating Thread-Safe Classes Automatically	1
86	Towards Interpreting Recurrent Neural Networks through Probabilistic Abstraction	
87	Trace-Checking Signal-based Temporal Properties: A Model-Driven Approach	1
88	UI Obfuscation and Its Effects on Automated UI Analysis for Android Apps	
89	UnchartIt: An Interactive Framework for Program Recovery from Charts	1
90	Understanding Performance Concerns in the API Documentation of Data Science Libraries	
91	Verified from Scratch: Program Analysis for Learners' Programs	1
92	Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks	1
93	Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling	1
94		55
95	code-related	59,14%

ICSE 2020

RESEARCH

Knowing a bit about statistics, I took it a bit further and quickly labeled all of the papers by just the title and the abstract.

The ratio was about the same: most of the papers are dedicated to code in one way or another.

Argument 1: we are biased towards coding

117	Time-travel Testing of Android AppsTechnical Zhen Dong, Marcel Böhme, Lucia Cojocaru, Abhik Roychoudhury	1
118	Towards Characterizing Adversarial Defects of Deep Learning Software from the Lens of U Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, Meng Su Pre-print	1
119	Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Zishuo Ding, Jinfu Chen, Weiyi Shang Pre-print	1
120	Translating Video Recordings of Mobile App Usages into Replayable ScenariosTechnical Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, Pre-print Media Attached	
121	Typestate-Guided Fuzzer for Discovering Use-after-Free VulnerabilitiesTechnical Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongx Link to publication DOI Pre-print	1
122	Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by De Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, Jinsl	
123	Understanding the Automated Parameter Optimization on Transfer Learning for Cross-Proj Ke Li, Zilin Xiang, Tao Chen, Shuo Wang, Kay Chen Tan Pre-print	1
124	Unsuccessful Story about Few Shot Malware-Family Classification and Siamese Network t Yude Bai, Zhenchang Xing, Li Xiaohong, Zhiyong Feng, Duoyuan Ma	1
125	Verifying Object ConstructionTechnical Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, Michael D. Ernst	1
126	Watchman: Monitoring Dependency Conflicts for Python Library EcosystemTechnical Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-C	1
127	When APIs are Intentionally Bypassed: An Exploratory Study of API WorkaroundsTechnica Maxime Lamothe, Weiyi Shang Pre-print	1
128	White-box Fairness Testing through Adversarial SamplingTechnical Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song	
129	sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart ContractsTechnical Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, Minh Quang Tran	1
130		72
131	code-related	55,81%

ASE 2020

RESEARCH

I did the same for ASE, with similar results

Argument 1: we are biased towards coding

31	On the Relationship between User Churn and Software IssuesMSR - Technical Paper Omar El Zarif, Daniel Alencar Da Costa, Safwat Hassan, Ying Zou Pre-print Media Attached	
32	PUMiner: Mining Security Posts from Developer Question and Answer Websites with PU LearningMSR - Technical Paper Triet Le Huynh Minh, David Hin, Roland Croft, Muhammad Ali Babar DOI Pre-print Media Attached	
33	Painting Flowers: Reasons for Using Single-State State Machines in Model-Driven EngineeringMSR - Technical Paper Nan Yang, Pieter Cuijpers, Ramon Schiffelers, Johan Lukkien, Alexander Serebrenik Media Attached	
34	Polyglot and Distributed Software Repository Mining with CROSSFLOWMSR - Technical Paper Konstantinos Barmpis , Patrick Neubauer, Jonathan Co, Dimitris Kolovos, Nicholas Matragkas, Richard Paige Media Attached	1
35	RTPTorrent: An Open-source Dataset for Evaluating Regression Test PrioritizationMSR - Technical Paper Toni Mattis, Patrick Rein, Falco Dürsch, Robert Hirschfeld DOI Pre-print Media Attached	1
36	SoftMon: A Tool to Compare Similar Open-source Software from a Performance PerspectiveMSR - Technical Paper Shubhankar Suman Singh, Smruti Ranjan Sarangi Pre-print Media Attached	
37	The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSLMSR - Technical Paper James Walden Pre-print Media Attached	
38	The Scent of Deep Learning Code: An Empirical StudyMSR - Technical Paper Hadhemi Jebnoun, Masud Rahman, Foutse Khomh, Houssem Ben Braiek Pre-print Media Attached	1
39	The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHubMSR - Technical Paper Danielle Gonzalez, Thomas Zimmermann, Nachiappan Nagappan DOI Pre-print Media Attached	
40	Traceability Support for Multi-Lingual Software ProjectsACM SIGSOFT Distinguished Paper AwardMSR - Technical Paper Yalin Liu, Jinfeng Lin, Jane Cleland-Huang Media Attached	1
41	Using Large-Scale Anomaly Detection on Code to Improve Kotlin CompilerMSR - Technical Paper Timofey Bryksin, Victor Petukhov, Ilya Alexin, Stanislav Prikhodko, Alexey Shpilman, Vladimir Kovalenko, Nikita Povarov Pre-print Media Attached	1
42	Using Others' Tests to Avoid Breaking UpdatesMSR - Technical Paper Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, Shane McIntosh Pre-print Media Attached	1
43	Visualization of Methods Changeability Based on VCS DataMSR - Technical Paper Sergey Svitkov, Timofey Bryksin Pre-print Media Attached	1
44	What constitutes Software? An Empirical, Descriptive Study of ArtifactsMSR - Technical Paper Rolf-Helge Pfeiffer Pre-print Media Attached	
45	What is the Vocabulary of Flaky Tests?MSR - Technical Paper Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, Antonia Bertolino Pre-print Media Attached	1
46		27
47		60,00%

MSR 2020

RESEARCH

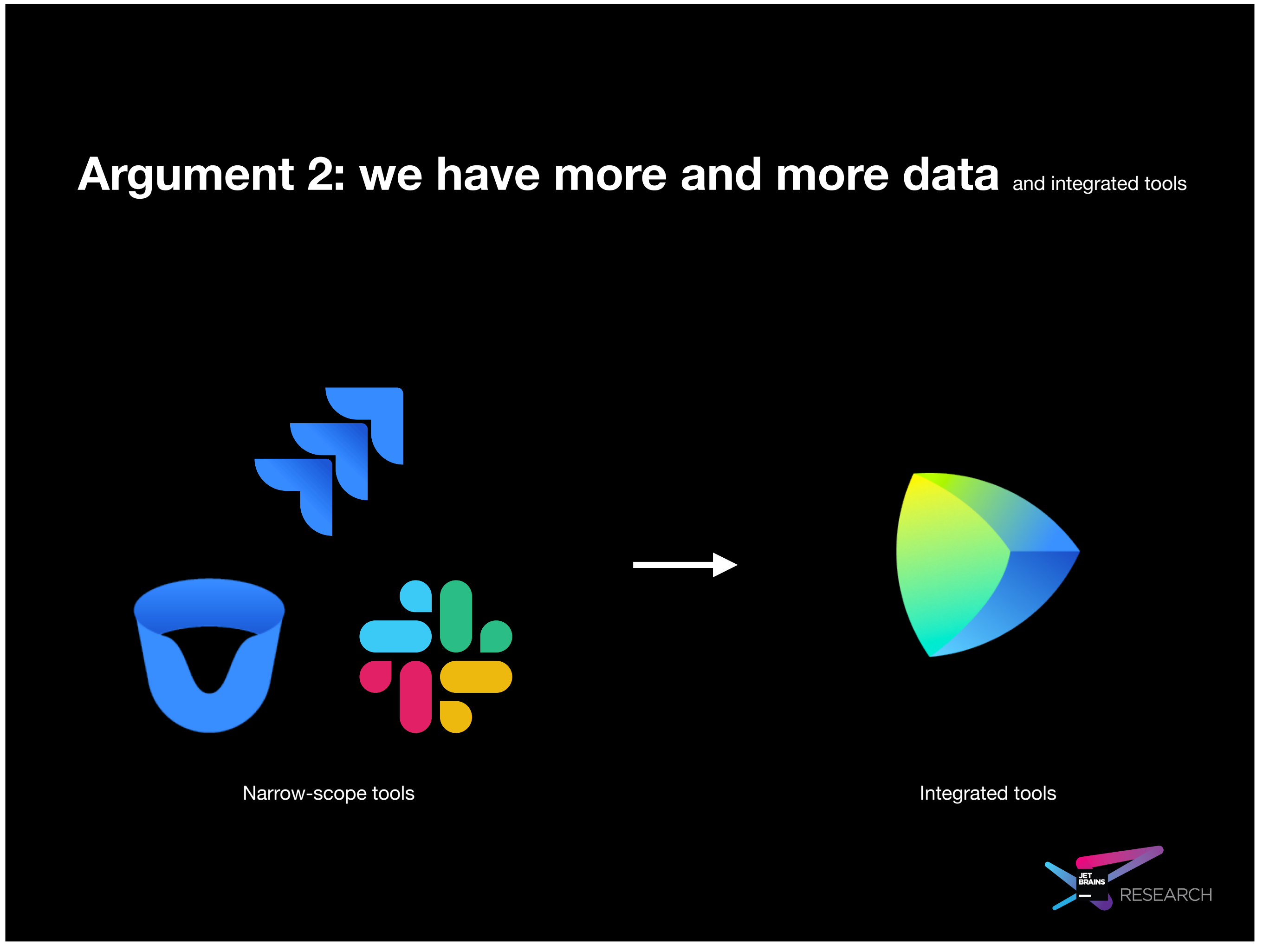
And for MSR. Same.

For some reason, it feels like the SE research community considers code — and its analysis and manipulation — more important than activities besides coding.

Argument 2: we have more and more data

RESEARCH

Another argument is, the tools are becoming more integrated. A typical situation is, a team tracks issues in Jira, chats in Slack, does code reviews in Crucible, also hosts code in a dedicated service, runs CI builds in yet another dedicated service…
Teams often have to maintain a whole zoo of tools, and each of those tools is normally isolated from others. If we want to share the data between tools, the amount of integration work grows quadratically as the number of tools grows.



Lately, more and more of these activities have been supported by what I call platform tools, such as JetBrains Space or Github. The cool part here is, we can use data from all collaborative activities in one context basically for free!
This enables us to benefit from synergy between more and more data types, so more and more data-driven enhancements are possible.

Moreover, modern collaboration tools are more platforms than tools: they are very friendly to extension.

How do we improve collaboration tools?

- Observability
- Decision-making support
- Taming the chaos

RESEARCH

OK, now that you are hopefully convinced that collaboration tools are worth special attention, let me share just a few broad directions of how exactly we could make the collaboration tools of tomorrow smarter and more helpful.

Here are a few broad categories of approaches: [list]

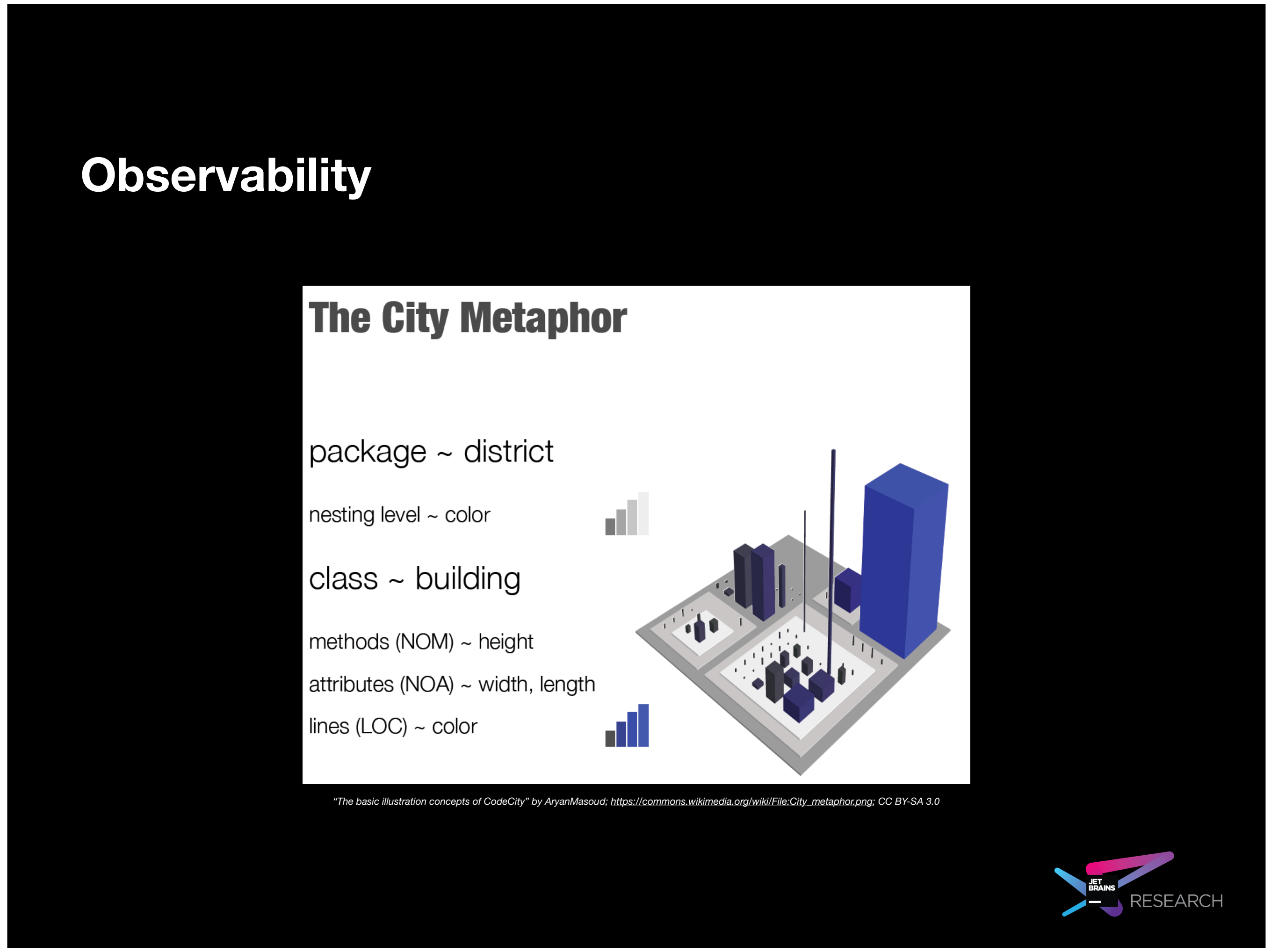
Now let’s look into each of these categories.

Observability

RESEARCH

Well, observability is a relatively straightforward idea. Development processes are complex, teams could be huge, it might be hard to make sense of what is going on — tools could help.

One obvious example of an observability feature is the Github's punchcard. With just a single glance, one can make sense of the user's activity through the year. While this is no rocket science, extracting this information would otherwise require careful processing of all repositories, so it certainly helps with a bird's eye view.



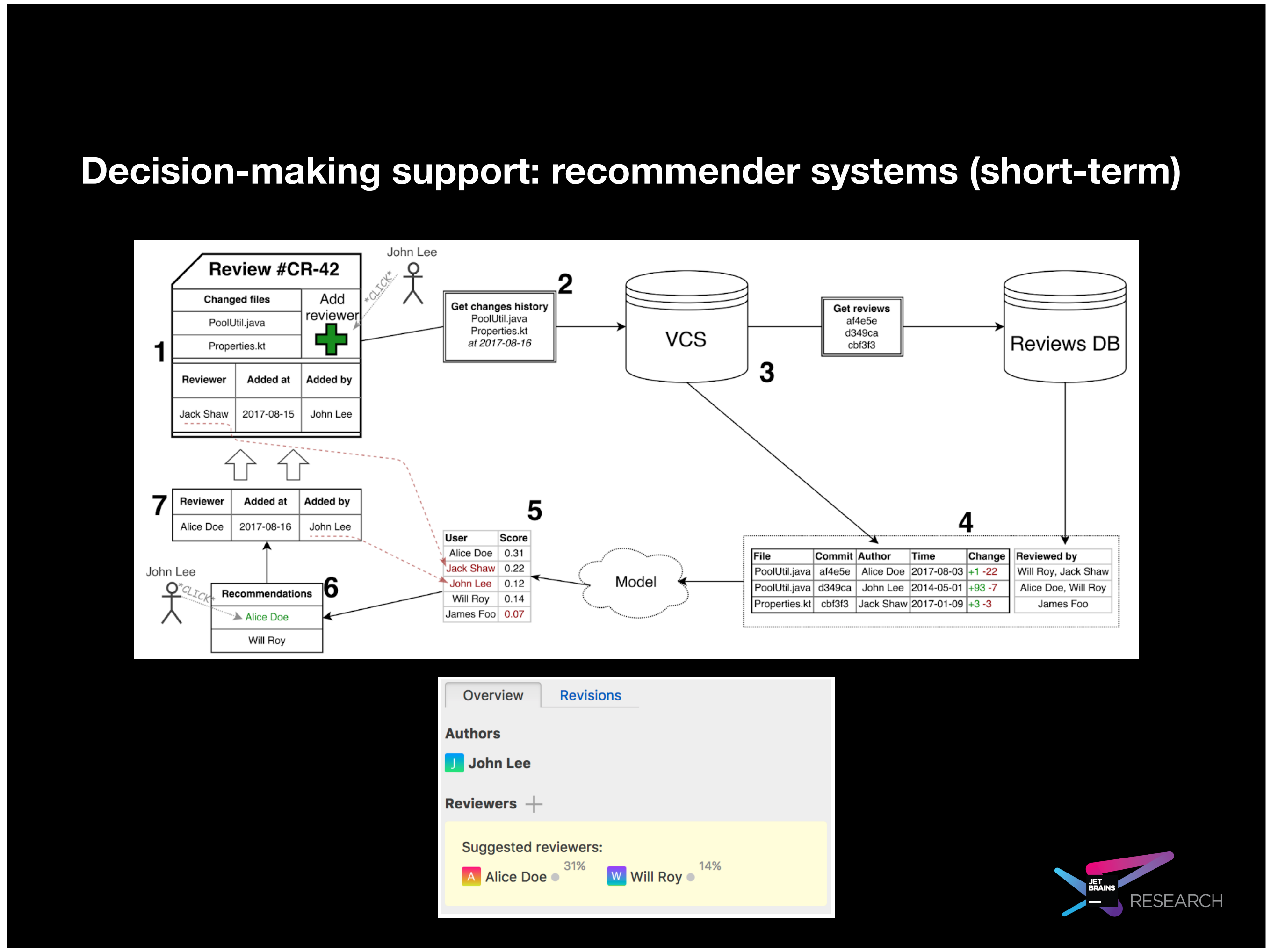
A more interesting example is the CodeCity concept, a rather famous one. If we visualise not just aggregated activity, but code attributes, this gives us a bird's eye view over the codebase, which could be really helpful with making decisions — for example, if there are any monster classes to refactor.

The point that I'd like to hightlight, we don't really have the major collaboration tools provide that much of a bird's eye view or other observability related to the engineering artifacts — and other important things, such as team communication.
Some tools have some reports, but they are normally really basic. Project management software is a bit ahead here with Gantt charts and alike, but that's pretty much it.
Dedicated tools like CodeScene by Adam Tornhill and his team are still very niche instruments.

In our team, we are working to enable top-notch observability not just for technical artifacts, but also for the communication processes and collaborative work.
This is a really challenging task, I'm going to mention some of the challenges a bit later.

Decision-making support

Another thing that collaboration tools could be better at, is provide decision-making support.

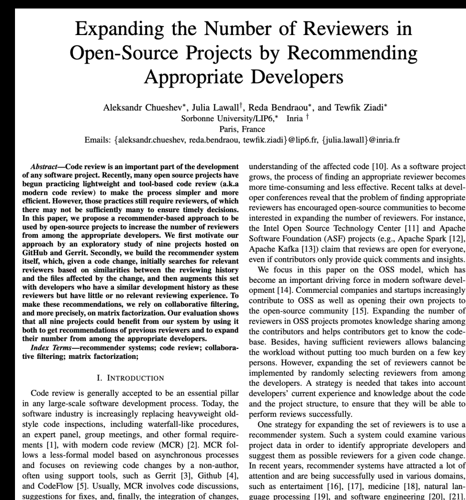


My favourite example here is code reviewer recommendation, I’ve worked on this problem a lot during my PhD and even before that. Your code review tool knows about all your contributions to the project, so it could help find the expert for a new review. Luckily, it’s already a really mainstream feature.

There are also other contexts for recommenders in collaboration tools: for example, automatic bug assignment algorithms are quite similar to reviewer recommendation, and are also potentially useful in their own way.

However, there are some issues: if we rely too much on such recommenders, it’s mostly the experts who are going to gain more expertise, so such recommenders may promote inequality in knowledge distribution, which is not good.

Decision-making support: recommender systems (long-term)



Expanding the Number of Reviewers in Open-Source Projects by Recommending Appropriate Developers

Aleksandr Chueshev*, Julia Lawall†, Reda Bendraou*, and Tewfik Ziadi*
Sorbonne University/LIP6,* Inria †
Paris, France
Emails: {aleksandr.chueshev, reda.bendraou, tewfik.ziadi}@lip6.fr, {julia.lawall}@inria.fr

***Abstract*—Code review is an important part of the development of any software project. Recently, many open source projects have begun practicing lightweight and tool-based code review (a.k.a modern code review) to make the process simpler and more efficient. However, those practices still require reviewers, of which there may not be sufficiently many to ensure timely decisions. In this paper, we propose a recommender-based approach to be used by open-source projects to increase the number of reviewers from among the appropriate developers. We first motivate our approach by an exploratory study of nine projects hosted on GitHub and Gerrit. Secondly, we build the recommender system itself, which, given a code change, initially searches for relevant reviewers based on similarities between the reviewing history and the files affected by the change, and then augments this set with developers who have a similar development history as these reviewers but have little or no relevant reviewing experience. To make these recommendations, we rely on collaborative filtering, and more precisely, on matrix factorization. Our evaluation shows that all nine projects could benefit from our system by using it both to get recommendations of previous reviewers and to expand their number from among the appropriate developers.**

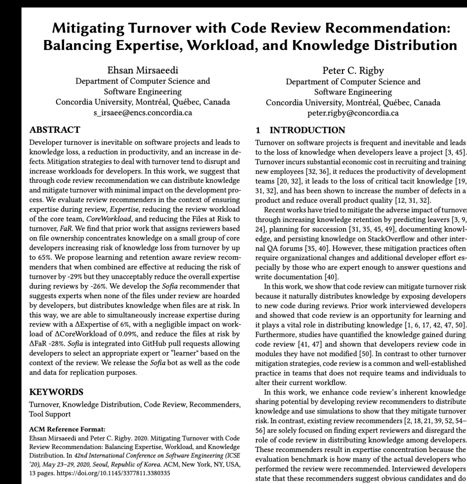
***Index Terms*—recommender systems; code review; collaborative filtering; matrix factorization;**

I. Introduction

Code review is generally accepted to be an essential pillar in any large-scale software development process. Today, the software industry is increasingly replacing heavyweight old-style code inspections, including waterfall-like procedures, an expert panel, group meetings, and other formal requirements [1], with modern code review (MCR) [2]. MCR follows a less-formal model based on asynchronous processes and focuses on reviewing code changes by a non-author, often using support tools, such as Gerrit [3], Github [4], and CodeFlow [5]. Usually, MCR involves code discussions, suggestions for fixes, and, finally, the integration of changes, understanding of the affected code [10]. As a software project grows, the process of finding an appropriate reviewer becomes more time-consuming and less effective. Recent talks at developer conferences reveal that the problem of finding appropriate reviewers has encouraged open-source communities to become interested in expanding the number of reviewers. For instance, the Intel Open Source Technology Center [11] and Apache Software Foundation (ASF) projects (e.g., Apache Spark [12], Apache Kafka [13]) claim that reviews are open for everyone, even if contributors only provide quick comments and insights.

We focus in this paper on the OSS model, which has become an important driving force in modern software development [14]. Commercial companies and startups increasingly contribute to OSS as well as opening their own projects to the open-source community [15]. Expanding the number of reviewers in OSS projects promotes knowledge sharing among the contributors and helps contributors get to know the code-base. Besides, having sufficient reviewers allows balancing the workload without putting too much burden on a few key persons. However, expanding the set of reviewers cannot be implemented by randomly selecting reviewers from among the developers. A strategy is needed that takes into account developers' current experience and knowledge about the code and the project structure, to ensure that they will be able to perform reviews successfully.

One strategy for expanding the set of reviewers is to use a recommender system. Such a system could examine various project data in order to identify appropriate developers and suggest them as possible reviewers for a given code change. In recent years, recommender systems have attracted a lot of attention and are being successfully used in various domains, such as entertaiment [16], [17], medicine [18], natural language processing [19], and software engineering [20], [21].



Mitigating Turnover with Code Review Recommendation: Balancing Expertise, Workload, and Knowledge Distribution

Ehsan Mirsaeedi
Department of Computer Science and Software Engineering
Concordia University, Montréal, Québec, Canada
s_irsaee@encs.concordia.ca

Peter C. Rigby
Department of Computer Science and Software Engineering
Concordia University, Montréal, Québec, Canada
peter.rigby@concordia.ca

ABSTRACT

Developer turnover is inevitable on software projects and leads to knowledge loss, a reduction in productivity, and an increase in defects. Mitigation strategies to deal with turnover tend to disrupt and increase workloads for developers. In this work, we suggest that through code review recommendation we can distribute knowledge and mitigate turnover with minimal impact on the development process. We evaluate review recommenders in the context of ensuring expertise during review, *Expertise*, reducing the review workload of the core team, *CoreWorkload*, and reducing the Files at Risk to turnover, *FaR*. We find that prior work that assigns reviewers based on file ownership concentrates knowledge on a small group of core developers increasing risk of knowledge loss from turnover by up to 65%. We propose learning and retention aware review recommenders that when combined are effective at reducing the risk of turnover by -29% but they unacceptably reduce the overall expertise during reviews by -26%. We develop the *Sofia* recommender that suggests experts when none of the files under review are hoarded by developers, but distributes knowledge when files are at risk. In this way, we are able to simultaneously increase expertise during review with a ΔExpertise of 6%, with a negligible impact on workload of ΔCoreWorkload of 0.09%, and reduce the files at risk by ΔFaR -28%. *Sofia* is integrated into GitHub pull requests allowing developers to select an appropriate expert or "learner" based on the context of the review. We release the *Sofia* bot as well as the code and data for replication purposes.

KEYWORDS

Turnover, Knowledge Distribution, Code Review, Recommenders, Tool Support

ACM Reference Format:
Ehsan Mirsaeedi and Peter C. Rigby. 2020. Mitigating Turnover with Code Review Recommendation: Balancing Expertise, Workload, and Knowledge Distribution. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380335

1 INTRODUCTION

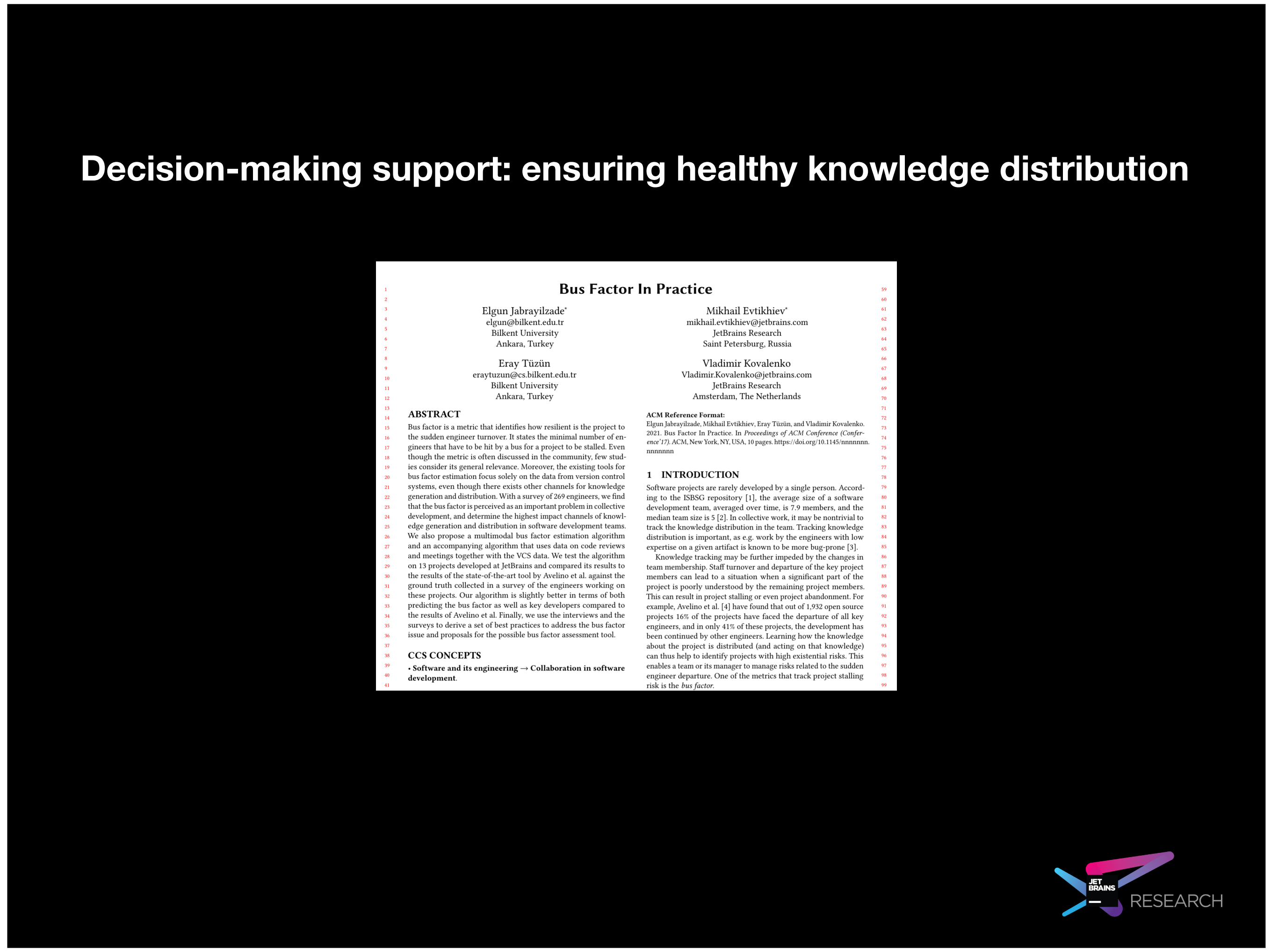
Turnover on software projects is frequent and inevitable and leads to the loss of knowledge when developers leave a project [3, 45]. Turnover incurs substantial economic cost in recruiting and training new employees [32, 36], it reduces the productivity of development teams [20, 32], it leads to the loss of critical tacit knowledge [19, 31, 32], and has been shown to increase the number of defects in a product and reduce overall product quality [12, 31, 32].

Recent works have tried to mitigate the adverse impact of turnover through increasing knowledge retention by predicting leavers [3, 9, 24], planning for succession [31, 35, 45, 49], documenting knowledge, and persisting knowledge on StackOverflow and other internal QA forums [35, 40]. However, these mitigation practices often require organizational changes and additional developer effort especially by those who are expert enough to answer questions and write documentation [40].

In this work, we show that code review can mitigate turnover risk because it naturally distributes knowledge by exposing developers to new code during reviews. Prior work interviewed developers and showed that code review is an opportunity for learning and it plays a vital role in distributing knowledge [1, 6, 17, 42, 47, 50]. Furthermore, studies have quantified the knowledge gained during code review [41, 47] and shown that developers review code in modules they have not modified [50]. In contrast to other turnover mitigation strategies, code review is a common and well-established practice in teams that does not require teams and individuals to alter their current workflow.

In this work, we enhance code review's inherent knowledge sharing potential by developing review recommenders to distribute knowledge and use simulations to show that they mitigate turnover risk. In contrast, existing review recommenders [2, 18, 21, 39, 52, 54–56] are solely focused on finding expert reviewers and disregard the role of code review in distributing knowledge among developers. These recommenders result in expertise concentration because the evaluation benchmark is how many of the actual developers who performed the review were recommended. Interviewed developers state that these recommenders suggest obvious candidates and do

But what if we keep the potential risks in mind? There are a couple pretty recent and really nice papers with a different take on reviewer recommendation: they propose focusing on workload, knowledge distribution, and diversity of the participants besides expertise.
To my knowledge, this kind of recommenders is yet to make its way into mainstream tools. My colleagues and I are also working in this direction.



Bus Factor In Practice

Elgun Jabrayilzade*
elgun@bilkent.edu.tr
Bilkent University
Ankara, Turkey

Mikhail Evtikhiev*
mikhail.evtikhiev@jetbrains.com
JetBrains Research
Saint Petersburg, Russia

Eray Tüzün
eraytuzun@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

Vladimir Kovalenko
Vladimir.Kovalenko@jetbrains.com
JetBrains Research
Amsterdam, The Netherlands

ABSTRACT

Bus factor is a metric that identifies how resilient is the project to the sudden engineer turnover. It states the minimal number of engineers that have to be hit by a bus for a project to be stalled. Even though the metric is often discussed in the community, few studies consider its general relevance. Moreover, the existing tools for bus factor estimation focus solely on the data from version control systems, even though there exists other channels for knowledge generation and distribution. With a survey of 269 engineers, we find that the bus factor is perceived as an important problem in collective development, and determine the highest impact channels of knowledge generation and distribution in software development teams. We also propose a multimodal bus factor estimation algorithm and an accompanying algorithm that uses data on code reviews and meetings together with the VCS data. We test the algorithm on 13 projects developed at JetBrains and compared its results to the results of the state-of-the-art tool by Avelino et al. against the ground truth collected in a survey of the engineers working on these projects. Our algorithm is slightly better in terms of both predicting the bus factor as well as key developers compared to the results of Avelino et al. Finally, we use the interviews and the surveys to derive a set of best practices to address the bus factor issue and proposals for the possible bus factor assessment tool.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**.

ACM Reference Format:
Elgun Jabrayilzade, Mikhail Evtikhiev, Eray Tüzün, and Vladimir Kovalenko. 2021. Bus Factor In Practice. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software projects are rarely developed by a single person. According to the ISBSG repository [1], the average size of a software development team, averaged over time, is 7.9 members, and the median team size is 5 [2]. In collective work, it may be nontrivial to track the knowledge distribution in the team. Tracking knowledge distribution is important, as e.g. work by the engineers with low expertise on a given artifact is known to be more bug-prone [3].

Knowledge tracking may be further impeded by the changes in team membership. Staff turnover and departure of the key project members can lead to a situation when a significant part of the project is poorly understood by the remaining project members. This can result in project stalling or even project abandonment. For example, Avelino et al. [4] have found that out of 1,932 open source projects 16% of the projects have faced the departure of all key engineers, and in only 41% of these projects, the development has been continued by other engineers. Learning how the knowledge about the project is distributed (and acting on that knowledge) can thus help to identify projects with high existential risks. This enables a team or its manager to manage risks related to the sudden engineer departure. One of the metrics that track project stalling risk is the *bus factor*.

My colleagues and I recently looked at the concept of bus factor — pretty much the measure of knowledge inequality risks — from different fresh angles.
This work is still under review, so I'll only touch on a couple main points.

Decision-making support: ensuring healthy knowledge distribution

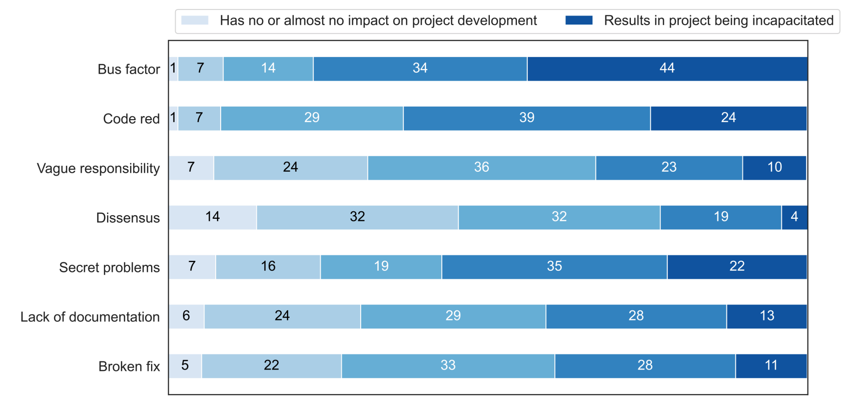


Figure 1: Perceived importance of the various collective development problems. All numbers are percentages

First, software professionals name unhealthy knowledge distribution as one of the most serious issues with collaborative development.

Decision-making support: ensuring healthy knowledge distribution

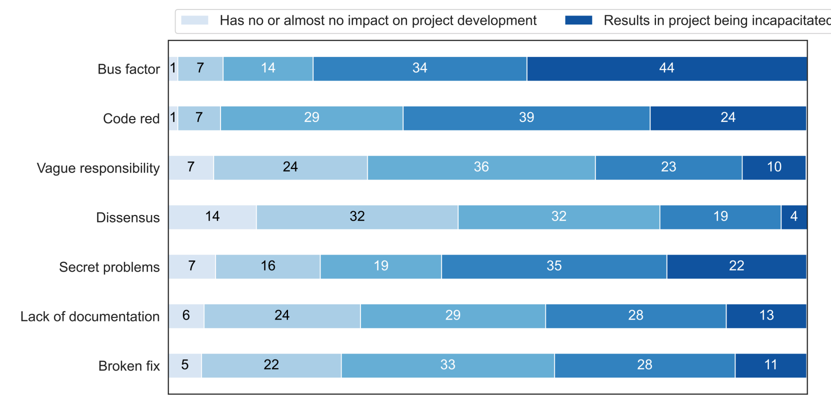


Figure 1: Perceived importance of the various collective development problems. All numbers are percentages

Most of our respondents would find a tool for bus factor calculation useful

RESEARCH

Second, most of them say that they would love for their tools to let them at least track the knowledge distribution, if not help maintain it.

Third, Additional data (e.g. code review history) provides better BF estimates

Decision-making support: ensuring healthy knowledge distribution

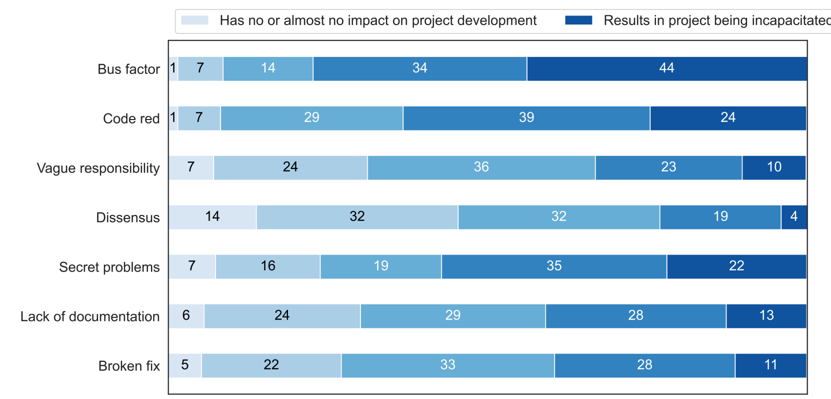


Figure 1: Perceived importance of the various collective development problems. All numbers are percentages

Most of our respondents would find a tool for bus factor calculation useful

Additional data (e.g. code review history) provides better BF estimates than just VCS

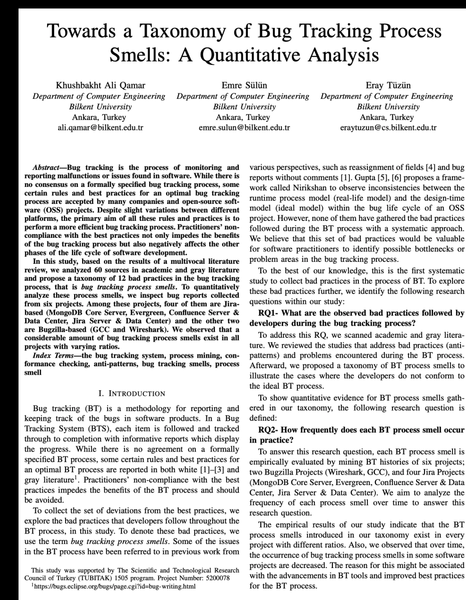
RESEARCH

Third, Additional data (e.g. code review history) provides better BF estimates than just code history

Taming the chaos: process smells

There is a popular concept of code smells — especially in the BENEVOL audience, I'm sure most of you at least heard about it. Some code smells can be located by static analysis and automatically fixed — that's one of the killer features of modern IDEs.

Taming the chaos: process smells



Towards a Taxonomy of Bug Tracking Process Smells: A Quantitative Analysis

Khushbakht Ali Qamar
Department of Computer Engineering
Bilkent University
Ankara, Turkey
ali.qamar@bilkent.edu.tr

Emre Sülün
Department of Computer Engineering
Bilkent University
Ankara, Turkey
emre.sulun@bilkent.edu.tr

Eray Tüzün
Department of Computer Engineering
Bilkent University
Ankara, Turkey
eraytuzun@cs.bilkent.edu.tr

***Abstract*—Bug tracking is the process of monitoring and reporting malfunctions or issues found in software. While there is no consensus on a formally specified bug tracking process, some certain rules and best practices for an optimal bug tracking process are accepted by many companies and open-source software (OSS) projects. Despite slight variations between different platforms, the primary aim of all these rules and practices is to perform a more efficient bug tracking process. Practitioners' non-compliance with the best practices not only impedes the benefits of the bug tracking process but also negatively affects the other phases of the life cycle of software development.**

In this study, based on the results of a multivocal literature review, we analyzed 60 sources in academic and gray literature and propose a taxonomy of 12 bad practices in the bug tracking process, that is *bug tracking process smells*. To quantitatively analyze these process smells, we inspect bug reports collected from six projects. Among these projects, four of them are Jira-based (MongoDB Core Server, Evergreen, Confluence Server & Data Center, Jira Server & Data Center) and the other two are Bugzilla-based (GCC and Wireshark). We observed that a considerable amount of bug tracking process smells exist in all projects with varying ratios.

***Index Terms*—the bug tracking system, process mining, conformance checking, anti-patterns, bug tracking smells, process smell**

I. Introduction

Bug tracking (BT) is a methodology for reporting and keeping track of the bugs in software products. In a Bug Tracking System (BTS), each item is followed and tracked through to completion with informative reports which display the progress. While there is no agreement on a formally specified BT process, some certain rules and best practices for an optimal BT process are reported in both white [1]–[3] and gray literature[1]. Practitioners' non-compliance with the best practices impedes the benefits of the BT process and should be avoided.

To collect the set of deviations from the best practices, we explore the bad practices that developers follow throughout the BT process, in this study. To denote these bad practices, we use the term *bug tracking process smells*. Some of the issues in the BT process have been referred to in previous work from various perspectives, such as reassignment of fields [4] and bug reports without comments [1]. Gupta [5], [6] proposes a framework called Nirikshan to observe inconsistencies between the runtime process model (real-life model) and the design-time model (ideal model) within the bug life cycle of an OSS project. However, none of them have gathered the bad practices followed during the BT process with a systematic approach. We believe that this set of bad practices would be valuable for software practitioners to identify possible bottlenecks or problem areas in the bug tracking process.

To the best of our knowledge, this is the first systematic study to collect bad practices in the process of BT. To explore these bad practices further, we identify the following research questions within our study:

RQ1- What are the observed bad practices followed by developers during the bug tracking process?

To address this RQ, we scanned academic and gray literature. We reviewed the studies that address bad practices (anti-patterns) and problems encountered during the BT process. Afterward, we proposed a taxonomy of BT process smells to illustrate the cases where the developers do not conform to the ideal BT process.

To show quantitative evidence for BT process smells gathered in our taxonomy, the following research question is defined:

RQ2- How frequently does each BT process smell occur in practice?

To answer this research question, each BT process smell is empirically evaluated by mining BT histories of six projects; two Bugzilla Projects (Wireshark, GCC), and four Jira Projects (MongoDB Core Server, Evergreen, Confluence Server & Data Center, Jira Server & Data Center). We aim to analyze the frequency of each process smell over time to answer this research question.

The empirical results of our study indicate that the BT process smells introduced in our taxonomy exist in every project with different ratios. Also, we observed that over time, the occurrence of bug tracking process smells in some software projects are decreased. The reason for this might be associated with the advancements in BT tools and improved best practices for the BT process.

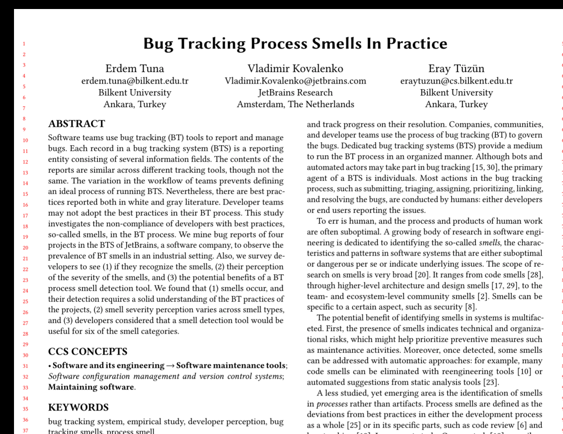
This study was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) 1505 program. Project Number: 5200078

[1]https://bugs.eclipse.org/bugs/page.cgi?id=bug-writing.html

But what if we go beyond code and search for smells in our processes?
Our colleagues from Bilkent University recently proposed a taxonomy of smells — potentially unwanted practices or patterns — in the bug tracking process.

This work merely proposes a framework: it doesn't elaborate on whether the smells are perceived as problems and whether people would like their tools to help avoid these smells.

Taming the chaos: process smells



Bug Tracking Process Smells In Practice

Erdem Tuna
erdem.tuna@bilkent.edu.tr
Bilkent University
Ankara, Turkey

Vladimir Kovalenko
Vladimir.Kovalenko@jetbrains.com
JetBrains Research
Amsterdam, The Netherlands

Eray Tüzün
eraytuzun@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

ABSTRACT

Software teams use bug tracking (BT) tools to report and manage bugs. Each record in a bug tracking system (BTS) is a reporting entity consisting of several information fields. The contents of the reports are similar across different tracking tools, though not the same. The variation in the workflow of teams prevents defining an ideal process of running BTS. Nevertheless, there are best practices reported both in white and gray literature. Developer teams may not adopt the best practices in their BT process. This study investigates the non-compliance of developers with best practices, so-called smells, in the BT process. We mine bug reports of four projects in the BTS of JetBrains, a software company, to observe the prevalence of BT smells in an industrial setting. Also, we survey developers to see (1) if they recognize the smells, (2) their perception of the severity of the smells, and (3) the potential benefits of a BT process smell detection tool. We found that (1) smells occur, and their detection requires a solid understanding of the BT practices of the projects, (2) smell severity perception varies across smell types, and (3) developers considered that a smell detection tool would be useful for six of the smell categories.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software configuration management and version control systems*; **Maintaining software**.

KEYWORDS

bug tracking system, empirical study, developer perception, bug tracking smells, process smell

and track progress on their resolution. Companies, communities, and developer teams use the process of bug tracking (BT) to govern the bugs. Dedicated bug tracking systems (BTS) provide a medium to run the BT process in an organized manner. Although bots and automated actors may take part in bug tracking [15, 30], the primary agent of a BTS is individuals. Most actions in the bug tracking process, such as submitting, triaging, assigning, prioritizing, linking, and resolving the bugs, are conducted by humans: either developers or end users reporting the issues.

To err is human, and the process and products of human work are often suboptimal. A growing body of research in software engineering is dedicated to identifying the so-called *smells*, the characteristics and patterns in software systems that are either suboptimal or dangerous per se or indicate underlying issues. The scope of research on smells is very broad [20]. It ranges from code smells [28], through higher-level architecture and design smells [17, 29], to the team- and ecosystem-level community smells [2]. Smells can be specific to a certain aspect, such as security [8].

The potential benefit of identifying smells in systems is multifaceted. First, the presence of smells indicates technical and organizational risks, which might help prioritize preventive measures such as maintenance activities. Moreover, once detected, some smells can be addressed with automatic approaches: for example, many code smells can be eliminated with reengineering tools [10] or automated suggestions from static analysis tools [23].

A less studied, yet emerging area is the identification of smells in *processes* rather than artifacts. Process smells are defined as the deviations from best practices in either the development process as a whole [25] or in its specific parts, such as code review [6] and bug tracking [18]. In a recent study, Qamar et al. [18] compile a

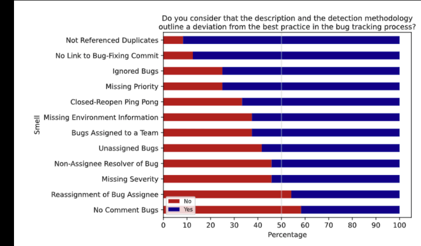


Figure 3: Cumulative representation of developers' view on the smell definition and its scope.

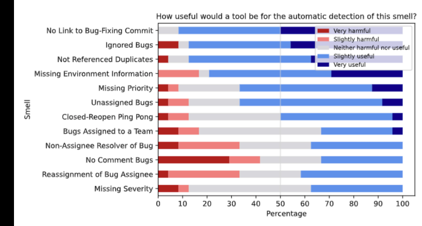


Figure 5: Usefulness rating of a potential tool with automatic smell detection feature.

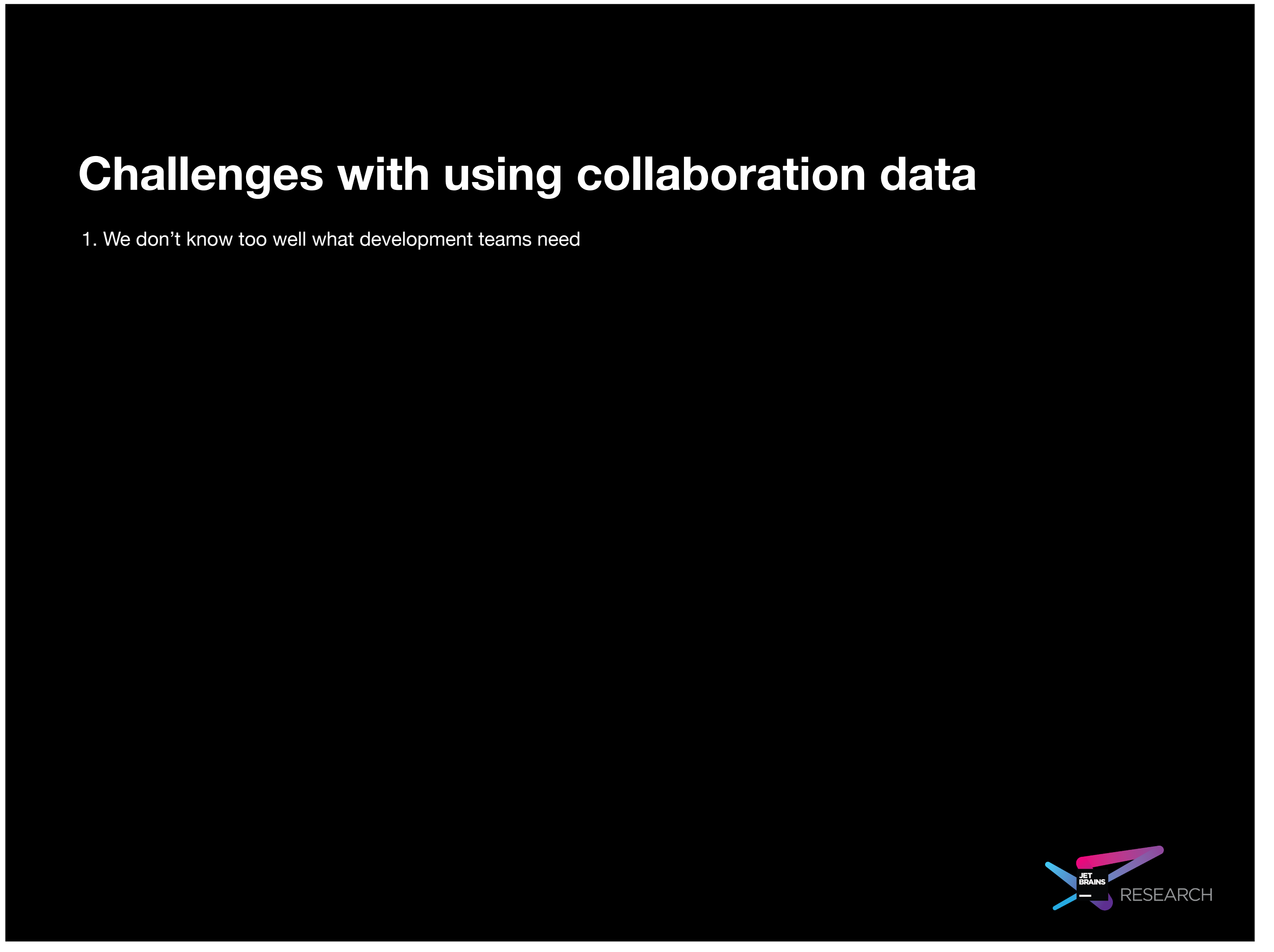
My colleagues and I took a close look at these smells this summer. Many of the smells are perceived as problems indeed; people say they would love for their tools to detect many of these smells.

JET BRAINS RESEARCH

Challenges with using collaboration data

RESEARCH

I've provided a few examples of directions for improvement of the tools. Now let me discuss a few challenges around this.

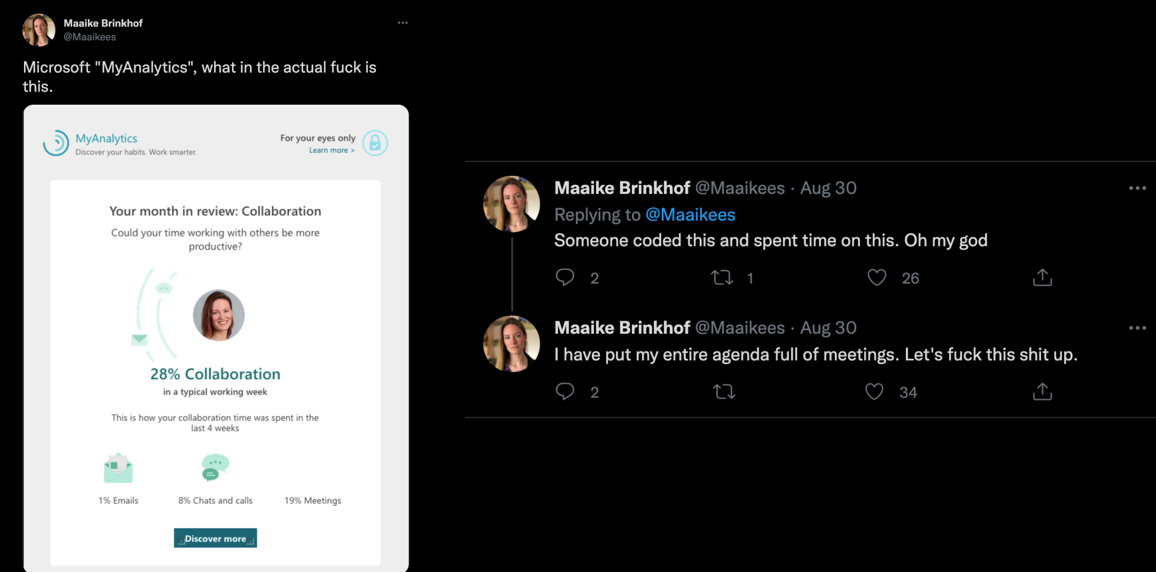


One more thing I didn't mention is that it would also be nice to actually try existing approaches in practice — think defect prediction, automated bug triaging, and such. We have more and more opportunities to test them in practice, and that is what we should be doing more.

This would help us address a big challenge: we researchers don't really know too well what people really need. There are plenty of different practices, traditions, methodologies, so many of the approaches are deemed to only be applicable in select scenarios.

Challenges with using collaboration data

2. Not everyone enjoys their data being processed, even if for their eyes only.



https://twitter.com/Maaikees/status/1432251254357495809

RESEARCH

No comment

Challenges with using collaboration data

3. Some do not enjoy it for a reason: there are actually quite a few ethical concerns

Xsolla reportedly lays off up to 150 people based on big data

CEO's communications cause controversy as staff deemed "not present" when working remotely are fired

Xsolla, a company that provides payments solutions and other services to the games industry, has reportedly laid off up to 150 people -- based primarily on a big data analysis of their productivity.

James Batchelor
Editor-in-Chief
Monday 9th August 2021

SHARE THIS ARTICLE

Recommend | Tweet | Share

https://www.gamesindustry.biz/articles/2021-08-09-xsolla-reportedly-lays-off-up-to-150-people-based-on-big-data

RESEARCH

No comment

Collaboration tools could be smarter!

And we could all help here.

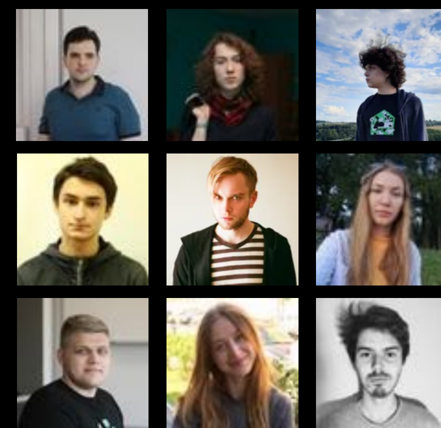
JET BRAINS RESEARCH

ICTL @ JetBrains Research

https://research.jetbrains.org/groups/ictl

I’m happy to say that I really practice what I preach.
I’d like to briefly introduce my team at JetBrains Research.
We are currently a team of 9, and we have a bunch of projects, mostly practice-oriented, where we seek to do exactly that: enable collaboration tools to better model their processes and provide assistance.

ICTL @ JetBrains Research



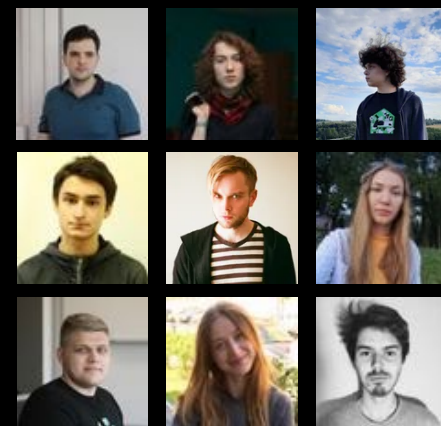
- Analysis of collaboration graphs (technical, social, combined)
- Expert recommendation systems on multimodal data
- Analysis of risky patterns of expertise distribution
- Duplicates search for SE
- Analysis of collaboration networks in the open source community

`https://research.jetbrains.org/groups/ictl`

RESEARCH

Here are the most prominent directions of our work. (list)

ICTL @ JetBrains Research



- Analysis of collaboration graphs (technical, social, combined)
- Expert recommendation systems on multimodal data
- Analysis of risky patterns of expertise distribution
- Duplicates search for SE
- Analysis of collaboration networks in the open source community

- **We are open to collaboration!**

`https://research.jetbrains.org/groups/ictl`

RESEARCH

We are open to collaborations with academic researchers!
Most of all, we like building tangible things, so if you like the idea of converting ideas to prototypes, and trying things in practice, please do reach out.

We are based in Amsterdam, Saint Petersburg, and Moscow, but we mostly work remotely.

[redacted]

Thank you!

vladimir.kovalenko@jetbrains.com

vovak.me

Finally, I'd like to thank you for your time and attention.

Thanks to organizers

Thank you for having me

Thank you!

vladimir.kovalenko@jetbrains.com
vovak.me