Enabling the Study of Software Development Behavior with Cross-Tool Logs

Ciera Jaspan, Matt Jorde, Carolyn Egelman, Collin Green, Ben Holtz, Edward Smith (Bloomberg), Maggie Hodges (Artech), Andrea Knight, Liz Kammer, Jill Dicker, Caitlin Sadowski, James Lin, Lan Cheng, Mark Canning, Emerson Murphy-Hill

Abstract

Understanding developers' day-to-day behavior can help answer important research questions, but capturing that behavior at scale can be challenging, particularly when developers use many tools in concert to accomplish their tasks. In this paper, we describe our experience creating a system that integrates log data from dozens of development tools at Google, including tools that developers use to email, schedule meetings, ask and answer technical questions, find code, build and test, and review code. The contribution of this article is a technical description of the system, a validation of it, and a demonstration of its usefulness.

KEYWORDS
D.2.8 Metrics/Measurement < D.2 Software Engineering < D Software/Software Engineering
INDEX TERMS
developer behavior, software engineering

Motivation

Humans are one of the most expensive parts of software creation. While companies can always add more equipment, adding more people comes with additional recruitment, training, and communication overhead. At Google, we have over 30,000 developers, so if we can make even small improvements to individual developers' productivity, we can attain large overall benefits for our products and their users.

At Google, the Engineering Productivity Research team's goal is to help our engineers be more productive. To meet this goal, our main piece of technical infrastructure is a quantitative logs pipeline that we call *InSession*. InSession helps us understand the behavior of developers by answering questions such as:

- Does a developer's certification in a programming language improve productivity?
- Can we predict negative interpersonal interactions between code authors and reviewers?

- Do new version control systems help developers be more productive?

In this article we describe InSession, which ingests logs from multiple developer tools to build a picture of a developer's behavior during their workday (see Supplementary Material for additional details). Like IDE monitors such as Mylyn Monitor [Murphy 2006] and ABB's system [Damevski 2016], InSession captures behavioral data automatically, but additionally captures data from tools outside of a developer's IDEs. Like multi-tool monitors such as HackyStat [Johnson 2007] and PROM [Sillitti 2003], InSession combines behavioral data from multiple developer tools, but in contrast to these systems, InSession ingests existing cloud-based tool logs, meaning that developer behavior is captured on every workstation without the need to install custom sensor software for each individual tool.

Privacy Principles

To maintain the integrity of data and trust of employees, strong privacy principles are critical. In consultation with privacy experts, we identified the following core principles in the form of 7 Dos and Don'ts:

1. **Do** ingest logs data for only employees. InSession ingests data from employees' use of the tools and systems provided by Google for them to do their work.
2. **Do** focus on tools used for work purposes**.** Avoid collecting task-identifying metadata from tools with mixed purposes such as email, but do collect such metadata for tools that are used exclusively for work purposes (e.g. code review).
3. **Don't** collect employee-generated content, with the exception of code repository paths, build rules, and similar artifacts that are available to the entire company.
4. **Do** encrypt stored data.
5. **Do** make stored data access auditable. We have an access logging system that tracks access to the raw files and their SQL-table views.
6. **Don't** report data for individual employees without the individual's prior consent. Individuals must not be identifiable in reported aggregate data. Some may be tempted to include this data in performance evaluations. However, we do not do so, both on principle and because research suggests developers are opposed to such metrics [Begel 2014].
7. **Do** destroy data after a set retention period. We use a retention period of 3 years for individual data which allows for year-over-year studies. Aggregated data is stored indefinitely.

Design and Implementation of InSession

Creating Events from Logs

We first discuss how InSession defines and classifies developer events. An *event* is a distinct usage of a tool or system by a developer or on a developer's behalf. Each log source has its own *importer* which extracts information into our common event data format. Events come from both developer-specific tools, like bug tracking, version control, and editors, and also from more general purpose tools, including Gmail and Calendar.

We distinguish between two types of event: *frontend* and *backend*. A frontend event is one that a developer actively initiates, e.g., clicking a UI button. A backend event is one that occurs asynchronously on a developer's behalf, e.g., cron jobs. Backend events are useful for performing studies related to what developers do after launching a long-running action, e.g., do they switch to another task or continue with related work while waiting? Events can also be either *instantaneous* or *durational*. An instantaneous event has an undefined endpoint, e.g., switching focus to a documentation page. A durational event has a set start and end time, e.g., running a build.

For most events, we also collect additional metadata about the event, which we call *artifacts*. These artifacts can be classified as *task-identifying* or *informational*. A task-identifying artifact is one that identifies a specific development task that can be used to group related events together, e.g., a development workspace label or identifier for a changelist (the proposed change to a codebase that generally undergoes review). An informational artifact provides contextual information about an event, e.g., the path of a file viewed in a code search tool. Each event importer determines how artifacts from its log source should be classified. For example, the changlist identifier for code review logs are always classified as task-identifying by its importer since it identifies the task of reviewing that changelist. The core artifacts we collect include changelist identifiers, development workspace labels, bug tracking identifiers, and code repository paths.

Creating Sessions from Events

To build a higher-level picture of engineering workflows, we organize groups of related events into *sessions*. Each session is designed to represent a contiguous block of time where the engineer works on a single task, such as coding or code review. Sessions are intended to represent active workflows, so we consider only frontend events when creating sessions. A session consists of a unique identifier, the ordered list of event identifiers included in the session, the developer behind those events, the union of the artifacts from those events, a start time which corresponds to the timestamp of the first included event, and an end time which corresponds to the timestamp of the last included event.

Figure 1 visualizes how events are combined into sessions. Sessions are created by grouping multiple events into single sessions when they happen on the same day, happen within some

time delta of each other (we use 10 minutes), and have the same task-identifying artifacts (or no task identifying artifact at all).

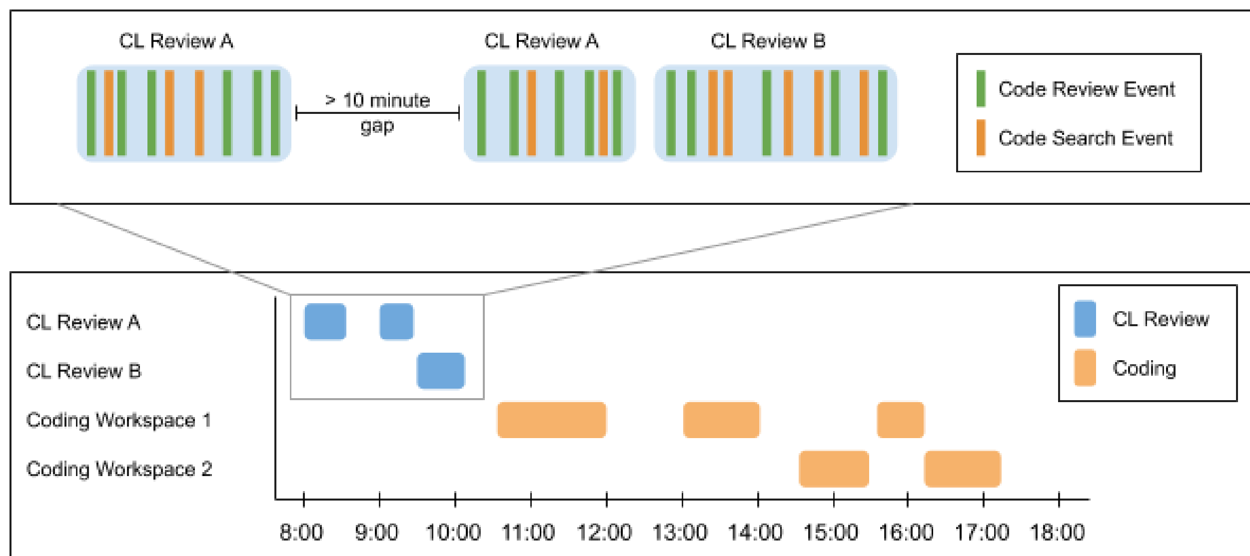


Figure 1. At top, an example of code review and code search tool events and how they are grouped into three sessions, based on the changelist (CL) they are associated with and the amount of time between them. At bottom, an example of an engineer's timeline of sessions split by task. Gaps can indicate, for example, breaks for lunch, unscheduled discussions, and tool usage for which we have not yet implemented log processing.

Creating Metrics from Sessions

Once events are organized into sessions, those sessions can be used to derive other metrics about developers' behavior. We selected seven metrics because they are ones that are useful in answering various questions about developer behavior and ones that our events could plausibly capture:

- Coding time, representing the time spent writing or maintaining code.
- Reviewing time, representing the time spent reviewing code.
- Shepherding time, representing the time spent addressing code review feedback.
- Investigation time, representing the time spent reading documentation.
- Development time, representing the time spent performing a development activity, of any type.
- Email time, the time spent interacting with email.
- Meeting time, the time spent in meetings.

InSession writes data in a format over which we can execute SQL queries, allowing for rapid analyses. For example, the following query examines total daily coding time before and after the declaration of COVID-19 as a global pandemic:

```
SELECT
  date,
  date >= DATE('2020-03-11') as is_during_pandemic,
  COUNT(DISTINCT employee) as num_working_employees,
  SUM(duration_micros) as total_coding_duration_per_day
FROM coding_time
GROUP BY date;
```

Validation Study

To use InSession with confidence in future studies, we performed a validation to understand the extent to which our metrics and behavioral self-reports of time use agree. Prior research has lamented the lack of validation in similar systems [Johnson 2007, §"Measurement and analysis validation"].

To obtain behavioral reports, we recruited 25 Google engineers to create diaries about what they did for a day, then compared their diaries against our sessions, both qualitatively and quantitatively using the PABAK agreement score [Byrt1993], which ranges from −1 (perfect disagreement) to 1 (perfect agreement).

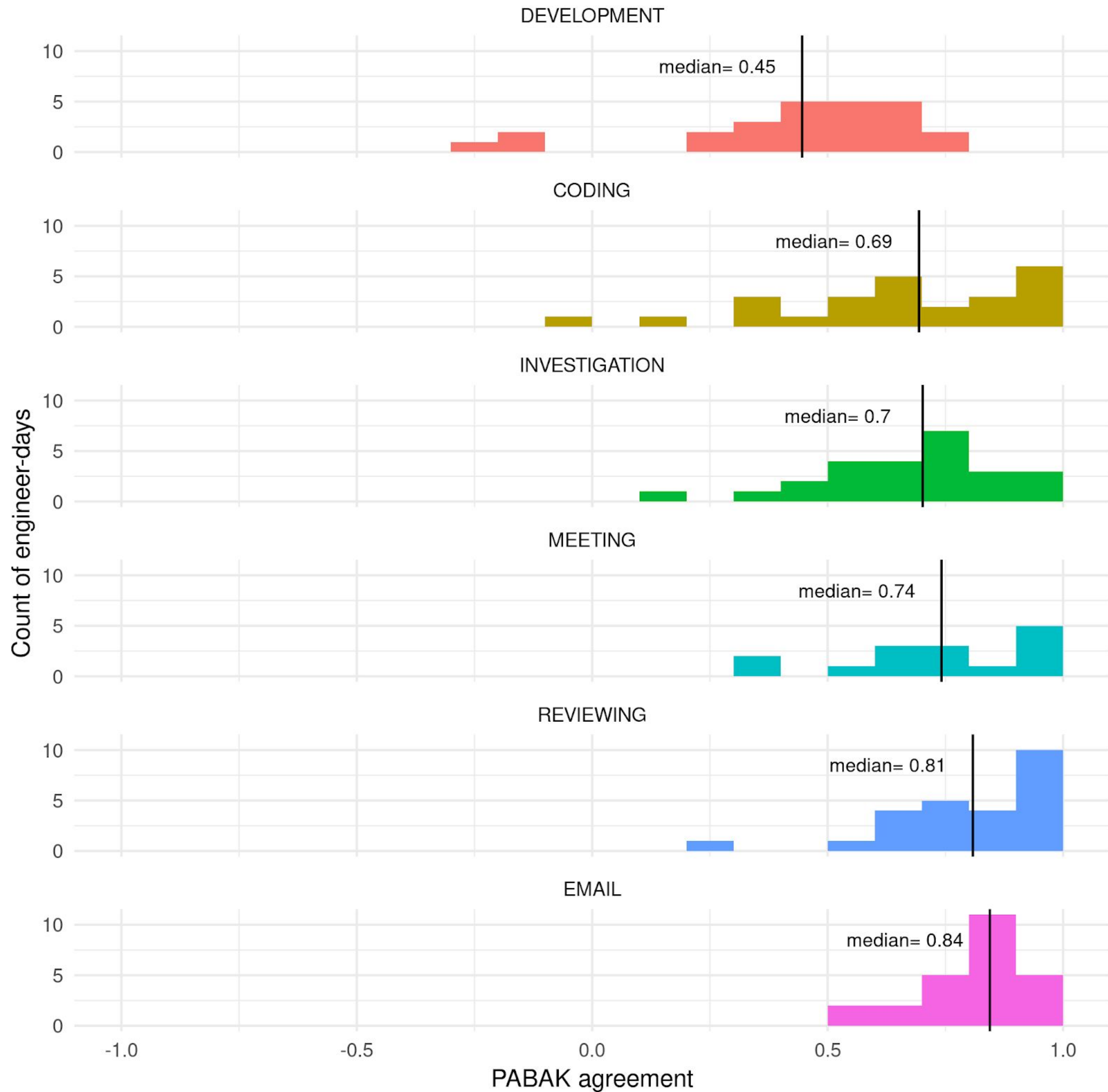


Figure 2. PABAK scores, segmented by activity type. Each data point in each histogram represents all of one participant's activities of that type in their day.

Figure 2 visualizes PABAK scores. To interpret PABAK scores, we use Allen and Yen's benchmarking approach called *norm-referencing* [Allen79]; since email (PABAK = 0.84) and meeting times (0.74[1]) are intuitively the simplest metrics, we treat their empirical agreement as *de facto* high, then compare all other metrics against this benchmark. By this measure,

---

[1] For meeting time, we excluded 10 participants because calendar data was unavailable for the day they completed the study. So *N*=15 for meeting time agreement, *N*=25 for all other agreement calculations. We return to the issue of missing data in our discussion.

reviewing time has high agreement (0.81), with coding (0.69) and investigation (0.70) nearly meeting this benchmark. Development time agreement (0.45) is well below this benchmark.

To understand mismatches between logs and diaries, we hand-analyzed data from the three participants for each session type who had the worst agreement scores among the 25 participants. Disagreements stemmed from four main sources: participants using tools that we do not yet have logs for, meetings that we did not have events for, participants multitasking, and participants forgetting to write tasks in their journals.

Overall, we interpret these results to mean that our reviewing, investigation, and coding time metrics are acceptably accurate, but further refinement to the development time metric is needed. The qualitative results suggest improvements to InSession, but also some fundamental limitations of capturing developer behavior through log data.

Application: The Impact of Readability

We next present a study illustrating the type of analysis that is possible with InSession.

In this study, we evaluate the effects of *readability*, a process of programming language certification at Google. Readability certifies that a developer understands best practices and coding style, for a specific programming language. We analyze C++ and Java readability, but readability exists for other languages.

Readability has been in use for many years at Google, but the benefits of readability were not clearly understood. We conducted a mixed-methods study to examine this, using our sessions data to examine two hypotheses, and surveys to provide context to those hypotheses. Our hypotheses are that, after a developer obtains readability certification:

- it will take reviewers less time to review their code, and
- it will take them less time to respond to reviewers' comments.

These hypotheses are based on the theory that if a developer has readability, their code will have fewer commonly encountered issues in code, resulting in reviewers not having to point these issues out, and the author not having to acknowledge and fix the issues.

To evaluate our hypotheses, we gathered code reviews of changelist authors who went through the readability process in a 10-month window, including their reviews before, during, and after the readability process. This included 104,947 code reviews for C++ and 99,614 code reviews for Java. We ran a linear regression that controlled for developer tenure, number of reviewers, and the size of the change. We additionally included a random effect for author identity, to control for characteristics of the individual developer.

The analysis shows the following effects ($p < .05$):

- An author having readability is associated with a review time that is lower by 4.5% for C++ and a non-statistically significant amount for Java, and
- An author having readability is associated with a shepherding time that is lower by 10.0% for Java and 10.5% for C++.

This quantitative data supports the hypotheses that readability has a positive effect for C++, with some support for Java. Survey data bears this out as well. 88% of engineers that completed the Java readability process said they agree with the statement “My readability experience was positive overall”, with 87% saying the same about the C++ readability process.

InSession can help investigate other questions about developer behavior. For instance, we recently found that we can predict negative interpersonal interactions during code review by using the 90th percentile of both reviewing and shepherding time [Cegelman 2020]. As another (unpublished) example, InSession helped us show that developers using a new version control system got their changes reviewed more quickly because the new tooling made it easier to create many small changes..

Finally, this study illustrates some of the limitations of InSession. For instance, we must assume that changes before and after the readability process are of equivalent quality; InSession itself provides no data to validate that assumption. Similarly, InSession does not say anything about whether engineers like the process, which is why we must collect complementary survey data.

Discussion

In this article, we have described the implementation, validation, and an application of InSession. Through building and maintaining InSession for more than three years, we have learned the following lessons:

- **Prioritize log sources**. Prioritize the inclusion of logs sources based on how useful each source is and on how easily it can be included. We prioritize logs from highly-adopted tools and those that are related to our research goals.
- **Enrich data as necessary**. We found that some log sources benefitted by enriching existing data; for example, adding gain and lose focus events to the code review tool logs increased the amount of measurable review activity by about 2 hours per week per engineer on average.
- **Validate data and metrics**. We found that the quality and consistency of data can differ between log sources, including unset or nonsensical values for data fields (e.g., negative durations), periods of missing or reduced data due to temporary system instability (e.g. this missing calendar data, described in our validation study), and outlier use cases (e.g. personal automation scripts). We also found that derived metrics, such as development time, while intuitive, can still disagree with human assessments. Thus, we recommend human validation like we describe in this article, but also automated monitoring and alerting for unexpected data during log ingestion.

While InSession isn't perfect, we have found it useful to understand developer behavior at scale. We envision a future where similar systems are deployed in other industrial and open source ecosystems, helping answer the most pressing questions about developers' behavior at work.

References

1. Murphy, G. et. al. (2006). How are Java software developers using the Eclipse IDE?. *IEEE Software*, *23*(4), 76–83.
2. Damevski, K. et al.. (2016). Mining sequences of developer interactions in visual studio for usage smells. *TSE*, 43(4), 359–371.
3. Johnson, P. (2007). Requirement and design trade-offs in Hackystat: An in-process software engineering measurement and analysis system. In *ESEM* (pp. 81–90).
4. Sillitti, A. et. al. (2003, September). Collecting, integrating and analyzing software metrics and personal software process data. In *EUROMICRO* (p. 336–343).
5. Begel, A., & Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. In *FSE* (pp. 12–23).
6. Byrt, T., et. al. (1993). Bias, prevalence and kappa. *Journal of Clinical Epidemiology*, 46(5), 423–429.
7. Allen, M., & Yen, W. (1979). Introduction to Measurement Theory. Monterey, CA: Brooks.
8. Egelman, C. D., et. al. (2020). Pushback: Characterizing and Detecting Negative Interpersonal Interactions in Code Review. In ICSE (to appear).

Author Bios

Ciera Jaspan is software engineer and manager at Google in Developer Intelligence. Her research interests are in empirical software engineering, mining software repositories, program analysis, and productivity metrics. She received her B.S. in Software Engineering from Cal Poly and her Ph.D. from Carnegie Mellon. She can be reached at ciera@google.com.

Matt Jorde is a software engineer at Google in Developer Intelligence. His research interests include software engineering topics such as human factors and software quality. He received a Master's degree in Computer Science from the University of Nebraska at Lincoln, and is a current member of the ACM. He can be reached at majorde@google.com.

Carolyn Egelman is a quantitative user experience researcher at Google in Developer Intelligence. Prior to Google she was a data science consultant. She earned her PhD at Carnegie Mellon in Engineering & Public Policy studying organizational learning in offshore manufacturing for high tech products. She can be reached at cegelman@google.com

Collin Green is a user experience researcher and manager at Google in Developer Intelligence. He received a B.Sc. in Psychology from the University of Oregon and a PhD in Psychology from the University of California, Los Angeles. He can be reached at colling@google.com.

Ben Holtz is a software engineer at Google in Developer Intelligence. He received a Master's degree in Computer Science from Stanford University. He can be reached at benholtz@google.com.

Edward Smith is a software engineer at Bloomberg. He received B.Sc. degrees in computer science and psychology from the University of Maryland College Park. He can be reached at esmith404@bloomberg.net

Maggie Hodges is a user experience researcher with Artech Information Systems at Google in Developer Intelligence. She has a Master’s of Public Health from University of California, Berkeley and previously worked at a consulting firm specializing in research related to program planning and evaluation for public sector clients. She can be reached at hodgesm@google.com

Andrea Knight is a user experience researcher at Google. She studied Information Systems and Human-Computer Interaction at Carnegie Mellon. Her research interests include applied product development research, video content selection processes, office and engineering productivity, and user privacy. She can be reached at aknight@google.com

Liz Kammer is a software engineer at Google. Her research interests are in software engineering, specifically in software developer diversity and inclusion. She received a Master’s degree in computer science from the University of Alabama. She can be contacted at eakammer@google.com.

Jill Dicker is a software engineer at Google in Developer Intelligence. She received a B.Sc. in mathematics and computer science from the University of British Columbia and a M.Sc in computer science from Simon Fraser University. She can be contacted at jdicker@google.com.

Caitlin Sadowski is a software engineer and manager at Google, where she aims to understand and improve developer workflows. Currently, she is helping Chrome developers make data-driven decisions as the manager of the Chrome Metrics & Analysis teams. In the past, she made static analysis useful at Google by creating the Tricorder program analysis platform. She has a PhD from the University of California, Santa Cruz where she worked on a variety of research topics related to programming languages, software engineering, and human computer interaction. She can be reached at supertri@google.com.

James Lin is a software engineer at Google in Developer Intelligence. His research interests include end-user programming and user interface design tools. James has a PhD in computer science from the University of California, Berkeley and previously worked at IBM Research. He is a member of ACM and IEEE. He can be reached at jameslin@google.com.

Lan Cheng is a quantitative user experience researcher at Google in Developer Intelligence. Her research interests are in measurement and impact evaluation of engineering productivity. She has a Ph.D. in economics at the University of California, Davis. She is a member of the Agricultural & Applied Economics Association. She can be contacted at lancheng@google.com.

Mark Canning is a software engineer at Google in Developer Intelligence. His research interests include engineering productivity and predictive modeling. He received the BA degree in Physics and in Mathematics from the University of California at Berkeley. He can be contacted at argusdusty@google.com.

Emerson Murphy-Hill is a research scientist at Google in Developer Intelligence. His research interests include software engineering and human-computer interaction. He received a B.Sc. at The Evergreen State College and a PhD in computer science at Portland State University. He can be contacted at emersonm@google.com.

Supplementary Material

Data Sources

InSession currently ingests employee event logs from the following development tool data sources:

- Buganizer, a bug tracking system
- g3doc, a markdown-based documentation system
- Code Search, a code searching and viewing tool
- Cider, a web-based IDE
- CitC, a cloud-based file system overlay on our monolithic code repository, which logs each save to a modified file during programming tasks;
- Blaze, a distributed build system
- Fig, a Mercurial-like source code management system
- Critique, a code review tool
- TAP, a continuous integration system
- YAQS, a Q&A system
- A build and test results viewer
- Three production monitoring tools
- Consolidated logs of command-line tool invocations, which cover about 90 command-line tools

Additionally, we include limited metadata from employees for three multi-purpose tools:

- Moma, an intranet search engine, for which we include only that a query was made and a boolean indicating whether it was a refinement of a previous query;
- Gmail, for which we include only timestamps of active use; and
- Google Calendar, for which we include only free/busy information and the number of invitees.

In total, InSession covers more than 100 different tools. A reader can find out more about these tools in the book *Software Engineering at Google* [Winters 2020].

Sessionization

The first step in creating sessions is to combine events from all log sources into a single stream, partition by developer, and order by timestamp (start time for durational events). Sessionization is then performed on these ordered partitions. For each partition, we first iterate over each event. The first event becomes the first *candidate session*. For each subsequent event, a decision is made whether to append it to the active candidate, or to begin a new candidate. Up to three candidate sessions can exist at any given time. The last one is considered active, and

the others remain pending until they are either merged or no longer eligible for merge as described in the next section. An event will be appended to the active candidate if:

1. the event and those of the active candidate occurred on the same day
2. the difference between the event's start time and that of the last event in the active candidate is within a predefined, heuristic time delta (we use 10 minutes); and
3. either of the following is true:
    - the event shares a task-identifying artifact with any event in the active candidate; or
    - the event has no task-identifying artifacts, and neither do any events in the active candidate.

An event that does not meet those criteria will begin a new active candidate. This process may result in two sessions for the same task, separated by a session with no task-identifying events, but which were nonetheless performed in support of the surrounding task. They can therefore be merged into a single session for the task.

Merging Sessions

To merge sessions, we iterate over the candidate sessions ordered by start time. Two sessions must be adjacent to be eligible for merge, i.e., one immediately follows the other, they occur on the same day, and they are separated by less than the time delta. Two adjacent sessions can be merged if they share at least one task-identifying artifact, or if one of them has no task-identifying artifacts and the other is the only one it is adjacent to. Three adjacent sessions can be merged if the first and last share at least one task-identifying artifact and the middle one has none. The session resulting from a merge contains the events from each session concatenated together, the union of their artifacts, and spans the time from the first session's start to the last session's end. Next, we describe this process in more detail.

Whenever an event begins a new active candidate session, the previous candidate session is placed in a queue of up to size 3, and a decision is made whether any candidates in the queue should be merged into a single session, or removed from the queue. We use '+' to represent the *merge* operation. The merge operation applied to two sessions $X+Y$ results in a new session Z with the following properties:

- The start time for Z is that of X
- The end time for Z is that of Y
- The events for Z are Y's events appended to X's; and
- The artifacts for Z are the union of those from X and Y

To help describe possible merge scenarios, we define the following notation:

- S_T : a candidate session that contains events with task-identifying artifacts
- S_{NT} : a candidate session that does not contain any events with task-identifying artifacts
- $X \cap Y$: the intersection of the set of session X's task-identifying artifacts, and the set of session Y's task-identifying artifacts
- *start_time(X)* : the start time of session X
- *end_time(X)* : the end time of session X

- $X+Y$: result of merging session X with session Y
- $X \rightarrow Y$: denotes adjacent sessions in the queue that are candidates for a possible merger

Two sessions X and Y are considered *adjacent* if Y immediately follows X, and they are within the time delta.

Only adjacent sessions can be merged. If a candidate is not adjacent to the existing candidates in the queue, those existing candidates are removed before adding the new candidate, so at any given time all candidates in the queue will be adjacent. The possible queue states and their merge decisions are enumerated below:

- $S_{NT} \rightarrow S_T$: $S_{NT}+S_T$ remains
- $S_T \rightarrow S_{NT} \rightarrow$ [time delay] : remove S_T+S_{NT}
- $S_{T1} \rightarrow S_{NT} \rightarrow S_{T2} \mid S_{T1} \cap S_{T2} \neq \varnothing$: $S_{T1}+S_{NT}+S_{T2}$ remains
- $S_{T1} \rightarrow S_{NT} \rightarrow S_{T2}$ | end_time(S_{T1}) = start_time(S_{NT}) : remove $S_{T1}+S_{NT}$, S_{T2} remains
- $S_{T1} \rightarrow S_{NT} \rightarrow S_{T2}$ | end_time(S_{NT}) = start_time(S_{T2}) : remove S_{T1}, $S_{NT}+S_{T2}$ remains
- $S_{T1} \rightarrow S_{NT} \rightarrow S_{T2} \mid S_{T1} \cap S_{T2} = \varnothing$: remove S_{T1} and S_{NT}, S_{T2} remains
- $S_{T1} \rightarrow S_{T2} \mid S_{T1} \cap S_{T2} = \varnothing$: remove S_{T1}, S_{T2} remains

In summary, if S_{NT} is the first or last of adjacent candidates, it is merged with the adjacent S_T. Otherwise, if S_{NT} appears between two S_Ts: all three are merged if the S_Ts contain overlapping task-identifying artifacts; S_{NT} is merged with one or the other if their bordering timestamps match; or each is removed from the queue individually, as a merge determination for S_{NT} could not be reached.

Creating Metrics from Sessions

- **Coding time**. The sum of the durations of sessions that contain file save events with a development workspace. Each changelist is associated with a workspace when it is committed, so InSession associates coding time with specific changelists even if the coding event happened before the changelist was created.
- **Reviewing time**. The sum of the durations of sessions that contain events from the code review tool such that the developer is assigned as a reviewer and the changelist is not yet submitted.
- **Shepherding time**. The sum of the durations of the author's sessions that contain events from the code review tool for a particular changelist, where such sessions must have started after the code review began but before the changelist was committed to version control. This represents the time an engineer is spending addressing comments on a code review.
- **Investigation time.** The sum of durations between two file save events, within the same session, where the engineer views documentation, runs search queries, or browses help forums. This represents the time the engineer needs to look up information while they are coding.
- **Development time.** The sum of durations of sessions where the engineer performs a development activity, of any type. The exceptions to development type are email and meeting time (below).

- **Email time.** The sum of durations of sessions where the engineer is accessing only email. We do not access any content from emails, and so we treat them as a single task for purposes of creating sessions. Email events are always clustered together in an email session, and any development event would start a new session. We acknowledge that engineers context switch as they read email and sometimes use email within the context of a development task.
- **Meeting time**. Meeting time is the time that engineers have accepted meetings in their calendars, excluding calendar items that are marked private (for privacy reasons), have only 1 attendee (not meetings, by definition), or have over 15 attendees (often "all-hands" meetings, where attendance may be low or non-participatory).

Other metrics may be possible in other contexts, such as planning time [Meyer2017] or program execution time [Astromskis2017]. From these metrics, we produce aggregations which can be queried outside our team while respecting individual developers' privacy. For example, an aggregation of the reviewing time metric may summarize the total amount of time spent reviewing code per week within company divisions.

Validation Study: Methodology

Note that neither the sessions nor the self-reports should be considered "correct" — both are likely to be inaccurate in different ways; sessions data, for instance, will be missing ad-hoc meetings, while self-reports will be missing data whenever participants forget or choose not to record their tasks. Instead, this comparison is an opportunity to understand how much, when, and why the subjective self-reports align and do not align with objective session data.

Participants. Twenty-five developers volunteered to participate from an invitation sent to Google employees that were randomly selected from a pool that:

- Reported using our main code repository in a quarterly survey;
- Are a software engineer, according to human resources data;
- Had a seniority of between entry-level and senior staff; and
- Had been at Google for 6 or more months.

Study Protocol. Participants completed the study in the following order:

- *Introduction*. Over 30 minutes, we provided participants with an overview of the study, obtained consent, and described the incentive structure. Incentives for participation were credits worth 65 USD or about 2 hours 15 minutes worth of massage. Partial incentives were awarded for partial study completion. We showed participants a completed time diary spreadsheet, as an example of the information they were supposed to collect.
- *Trial run*. Over 2-3 hours, participants practiced keeping a time diary in their own workplace. We instructed participants to:
    - Record what task they were performing at a given time, such as chatting with a colleague, writing interview feedback, or performing a code review;

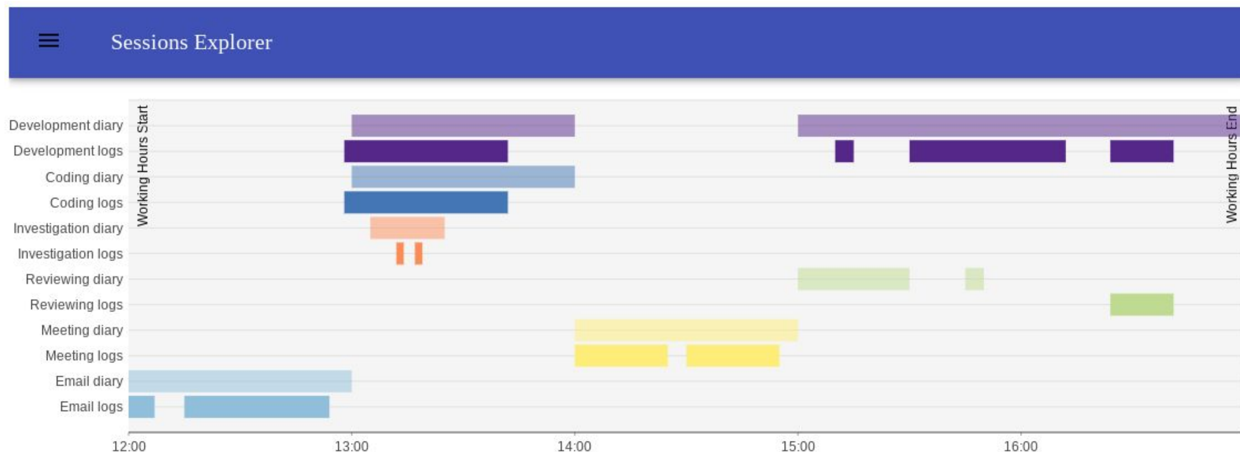
    - Use a tool to help record the time for the task, adding comments for context; and
    - Direct any questions to the researcher.

  The researcher gave feedback on the time diary, suggesting more or less detail as needed.
- *Data collection.* Over the course of one full workday, participants completed the time diary spreadsheet, recording their normal work activities. The researcher categorized each entry in the completed diary according to the metrics of interest in the study: development, coding, investigation, reviewing, meeting, and email times. The coded diaries were analyzed alongside logs to calculate agreement scores.
- *Debrief*. To understand diary and session mismatches, we held 30-minute debriefings with 13 participants whose diaries had the lowest agreement with logs. Each participant met with the researcher to review the time diary alongside a visualization of our log data superimposed with the coded diary data for the same day. Participants were instructed to talk through the events in the diary, confirm or correct how the researcher categorized the time, add notes to their diary, and comment about the visualization to help clarify why certain activities would have shown up in our logs. Agreement scores were re-calculated with any corrections made in the debrief session.

We performed two iterations of study pilots with 15 engineers to evaluate and refine our study protocol and facilitate planning for a full study. Only data from the full study is reported here.

Analysis. For quantitative analysis, we computed the agreement between sessions and diaries using Conger's Kappa [1985], an extension of Cohen's Kappa for continuous nominal scales, such as ours. We further adjust Conger's Kappa using Prevalence And Bias Adjustments (PABAK), as recommended by Byrt and colleagues [1993], so that agreement is not unduly impacted by very rare/common events or by data sources with very different criteria for indicating when events occur. For qualitative analysis, we analyzed participants' diaries and notes to understand when and why our logs did not match diary entries.

Validation Study: Agreement Example



To illustrate what agreement looks like, the figure above shows a pretend developer's workday, where the developer worked about 12:00 to 17:00. Each session metric is shown on the y-axis, and each metric is paired with the developer's diary entry. For instance, we see from the figure that the developer said that they were working on email continuously from 12:00 to 13:00, and our logs data roughly agrees with that, except that InSession divides email into two separate sessions. The PABAK agreement score for email is 0.91. In contrast, development time agreement is substantially lower, at 0.49. Reviewing and investigation agreement are in the middle with a PABAK scores of 0.65 and 0.89, respectively.

Validation Study: Qualitative Themes

- *Tools without logs*. Especially affecting coding and development sessions, some participants worked in development environments with logs that InSession does not currently ingest. Sometimes, these environments were from Alphabet companies that did not use the typical Google development environment. Adding breadth in the logs we ingest, such as from git or commercial IDEs, will help us address this shortcoming.
- *Missing meetings.* Participants reported being in meetings that our sessions did not capture, including ad-hoc, unscheduled, over-time, private, and large meetings. Ad-hoc and over-time meetings are a fundamental limitation of any software-based activity approach, and InSession's exclusion of private meetings was a design choice.
- *Multitasking*. Agreement was sometimes low because participants were performing multiple tasks at once, such as checking email during a meeting or having a conversation while tests were running. Heuristics informed by further research may increase the accuracy of our sessions.
- *Forgetting*. In several cases, participants simply forgot to write down task-relevant activities in their journals.

Readability Certification Process

To get readability certified, a candidate developer has to go through a process which varies between languages. Generally, the candidate's changelist undergoes Google's code review process [Sadowski 2018], but adds a randomly-selected reviewer from an evaluator queue. The queue contains developers already certified in the language in question. The evaluator provides feedback to the candidate about the changelist and completes a separate form that assesses the changelist. Readability certification is granted to the candidate once the assessment forms, collected over multiple reviews, demonstrate that the candidate has sufficient knowledge of the language.

Readability Study Plots

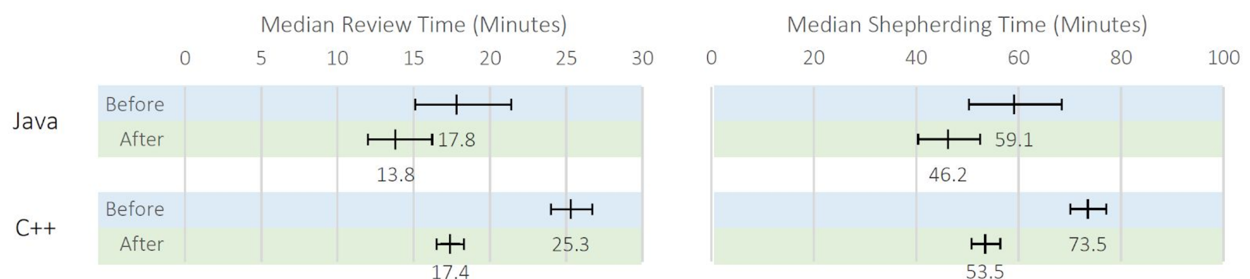


Figure 3. Median reviewing and shepherding times before and after developers obtained readability certification. Error bars represent 95% confidence intervals.

Overall, the trends in the table suggest that both hypotheses for Java and C++ are supported, because review and shepherding time decrease after readability certification. To improve the reliability of these estimates, we ran the regression shown in the main body of the paper.

References for Supplementary Material

1. Winters, T. et. al. (2020). Software Engineering at Google. Sebastopol, CA: O'Reilly Media.
2. Meyer, A. et. al (2017). The work life of developers: Activities, switches and perceived productivity. IEEE Transactions on Software Engineering, 43(12), 1178-1193.
3. Astromskis, S et. al. (2017). Patterns of developers behaviour: A 1000-hour industrial study. Journal of Systems and Software, 132, 85-97.
4. Conger, A. J. (1985). Kappa reliabilities for continuous behaviors and events. *Educational and Psychological Measurement, 45*(4), 861–868.
5. Sadowski, C., et. al. (2018, May). Modern code review: a case study at Google. In *ICSE-SEIP* (pp. 181–190).