Characterizing Software Maintenance Meetings: Information Shared, Discussion Outcomes, and Information Captured

Adriana Meza Soria
Taylor Lopez
Negin Mashhadi
André van der Hoek
amezasor@uci.edu
lopezta1@uci.edu
nmashhad@uci.edu
andre@uci.edu
Department of Informatics
University of California, Irvine
Irvine, CA, U.S.A.

Elizabeth Seero
Emily Evans
Janet Burge
l_seero@coloradocollege.edu
ea_evans@coloradocollege.edu
jburge@coloradocollege.edu
Department of Mathematics and Computer Science
Colorado College
Colorado, CO, U.S.A.

ABSTRACT

A type of meeting that has been understudied in the software engineering literature to date is what we term the software maintenance meeting: a regularly scheduled team meeting in which emergent issues are addressed that are usually out of scope of the daily stand-up but not necessarily challenging enough to warrant an entirely separate meeting. These meetings tend to discuss a wide variety of topics and are crucial in keeping software development projects going, but little is known about these meetings and how they proceed. In this paper, we report on a single exploratory case study in which we analyzed ten consecutive maintenance meetings from a major healthcare software provider. We analyzed what kind of information is brought into the discussions held in these meetings and how, what outcomes arose from the discussions, and what information was captured for downstream use. Our findings are varied, giving rise to both practical considerations for those conducting these kinds of meetings and new research directions toward further understanding and supporting them.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; *Software maintenance tools*; **Documentation**.

KEYWORDS

Meetings, software maintenance, information, resolution

ACM Reference Format:
Adriana Meza Soria, Taylor Lopez, Negin Mashhadi, André van der Hoek, Elizabeth Seero, Emily Evans, and Janet Burge. 2024. Characterizing Software Maintenance Meetings: Information Shared, Discussion Outcomes, and Information Captured. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623330

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623330

1 INTRODUCTION

Software systems spend a significant proportion of their lifetime undergoing maintenance [63]. For large deployed systems with a wide customer base, software teams need to keep track of how well the software is functioning at the various client sites, respond to any problems that arise, and plan, develop, and deploy enhancements [73]. Meetings play an essential role in performing this work. It is well-known that the weekly agenda of a typical developer may be dominated by meetings [31]. These meetings range in kind from daily stand-ups [81] and sprint planning [33], to dedicated design [61] and release planning [66], to impromptu [87] and retrospectives [34]. Together, the meetings in which a team engages represent an intricate network of activities and dependencies among them [79].

To date, meetings have been an understudied subject in the software engineering literature, which is a surprise given how frequent they are and given how much team-oriented intellectual work takes place in them that shapes the eventual product. Exceptions exist, with certain types of meetings that have been studied extensively, such Agile stand-ups (e.g., [80, 82, 83]) or whiteboard software design meetings (e.g., [20, 53, 75]). Specific aspects of meetings in general have also been examined in detail, such as meeting dynamics (e.g., [5, 51, 84]), inclusivity (e.g., [18, 45]), and the impact of hybrid and remote settings (e.g., [65, 67]).

This paper complements the existing literature on meetings in software engineering by focusing on software maintenance meetings [4]. To date, this kind of meeting has not been studied in the literature, but from the grey literature (e.g., [1, 30, 50, 52, 56, 70]), it is clear that it is a common type of meeting. The actual name for the meeting varies from maintenance meeting, to technical meeting, to weekly developer meeting, to weekly tech meeting, to engineering meeting, and more, but we favor the term maintenance meeting because a common trait is that the meetings take place in the context of an evolving existing system. What characterizes these meetings is that they: (1) serve an important role 'in between' the daily stand-ups many organizations employ and the dedicated, less frequent meetings in which a specific type of work gets done,

such as sprint planning or a whiteboard design meeting, (2) have a core set of participants consisting of the technical and project leads, with other team members attending as needed depending on the topic(s) to be discussed, (3) are regularly scheduled—usually weekly, sometimes more frequently—so that people can count on a forum for emergent issues that require forethought and discussion, and (4) address a broad range of issues. Maintenance meetings serve an important role in keeping software projects going, as numerous decisions are made regarding functionality of the software, complex issues with deployed systems are diagnosed, and team members can get help concerning non-trivial issues they face.

In preliminary work [4], we categorized the kinds of discussions that take place in maintenance meetings. Across ten meetings conducted by a single team from a major healthcare software provider, we found that the team engaged in forty-five discussions (termed *topics* in this paper) of fourteen different kinds, including assessing a problem, clarifying a misunderstanding, devising a solution, gathering knowledge, automating activities, performing a post-mortem, planning a future meeting agenda, reviewing a design proposal, and refining a ticket. This variety in the kinds of discussions held aligns with research into weekly meetings across industries, which found that weekly meetings tend to "discuss ongoing projects" and "routinely discuss the state of the business" [3].

Expanding upon our preliminary work, this paper seeks to deepen our understanding of maintenance meetings by examining how information creates a context for the discussions and outcomes in these kinds of meetings. How discussions in these meetings proceed depends on the availability of information, together with what information needs to be produced [9, 36, 42]. In other words, discussions in maintenance meetings on the one hand rely on information that is generated or available elsewhere and on the other hand produce information that shapes next steps on the project. We specifically address the following four research questions:

(1) *What kinds of information do developers rely on during maintenance meetings?*
(2) *How is this information brought into the meetings?*
(3) *What are the outcomes of maintenance meetings?*
(4) *Are these outcomes captured for future reference and, if so, how?*

The remainder of this paper is organized as follows. We detail the meetings upon which we perform our analysis in Section 2 and then introduce our methodology in Section 3. We present findings in Section 4 and discuss implications for research and practice in Section 5. We conclude with threats to validity in Section 6, related work in Section 7, and our plans for future work in Section 8.

2 DATA SET

As part of our prior work [4], we obtained copies of the WebEx recordings of ten maintenance meetings held at a major healthcare software development company. These maintenance meetings took place in the context of a variety of other meetings, including daily stand-ups, backlog refinement meetings twice a week, sprint planning every other week, retrospectives (infrequent), and a dedicated but temporary set of meetings related to a new UI initiative. The maintenance meetings are held by the *architecture committee*, a standing team that is responsible for maintaining and expanding a software system that is in use by hundreds of hospitals. The software stores terabytes of patient health data in the cloud and it is considered a critical system in the overall portfolio of the company. Meetings are always through WebEx, with some participants joining from the U.S.A. and the rest joining from India. Meetings are scheduled for one hour each, with some being slightly shorter and others going over a bit. The team addresses what it can in the hour and moves unaddressed items to the agenda for the week thereafter.

The meetings took place from March to July 2020. There were twelve different participants over that time period: a product owner and shadow product owner (O1, O2), two software architects (A1, A2), the lead quality assurance engineer (Q1), two managers (M1, M2), four developers (D1-D4), and an infrastructure engineer (I1). The main product owner, the two software architects, the QA engineer, and one of the managers were located in the U.S.A.; the rest of the participants were in India. A small core (O1, A1, A2, and Q1) attends nearly every meeting; others are only involved in the meetings either when a topic is discussed that they themselves placed on the agenda or when a topic requires their expertise. The median number of attended meetings by these other participants is 4.5. Table 1 documents precisely which participants attended which meetings.

Anecdotally and not necessarily by any metric being tracked, the architecture committee is considered a high-performing team by management, with management using this team and its practices as a model for organizing other development teams.

When we requested access to the WebEx meetings, we had several requirements. First, we wanted the meetings to be maintenance meetings and not of the other meeting types discussed. Second, we wanted the meetings to be consecutive over a period of time, to be able to identify potential issues around topics recurring (which we did not, but it was an objective in requesting consecutive meetings). Third, we requested ten meetings total, because it balances depth, in it being feasible to manually analyze ten hours of meetings in great detail, with breadth, in having several months of maintenance discussions available to examine and make sense of.

In our prior work, we found that the ten meetings covered forty-five distinct topics [4]. While each topic was unique, underneath were several shared objectives, ranging from assessing a problem, gathering knowledge, or devising a solution, to reviewing a design proposal, planning a future meeting agenda, or performing a post-mortem, to clarifying a misunderstanding, automating activities, or refining a ticket. Consider the following excerpt:

> **A2***: Do you see that?*
> **O1***: Oh, wow.*
> **A2***: So, what's on there? Zero instance. I just picked on that because it's 11 – Oh. That's – that's the reader. So, look at that. The CPU is higher on the read replica right now. Then if we look on their writer – Let's look at that over the last week.*

This excerpt is from a topic discussing the potential consequences of one of the clients onboarding additional users. In this case, as part of gathering knowledge to understand the ramifications of the client's plans, the team uses one of the tools in the AWS toolkit to study the CPU load live. In observing that it is higher on the read replica than on the write replica, they launch into a discussion as to why this may be (with the team looking up additional information

Table 1: Meeting participants.

Participant	Preferred pronoun	Role	Location	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
O1	she, her, hers	Product owner	U.S.A.	X	X	X	X	X	X	X			X
A1	he, him, his	Software architect	U.S.A.	X	X	X	X	X	X	X	X	X	X
A2	he, him, his	Software architect	U.S.A.	X	X	X	X	X	X	X	X	X	X
Q1	he, him, his	QA engineer	U.S.A.	X	X	X	X		X	X	X	X	
O2	she, her, hers	Product owner	India	X		X	X		X				X
M1	he, him, his	Manager	U.S.A.			X	X	X	X	X	X		
D1	he, him, his	Developer	India	X		X	X	X	X		X		X
D2	he, him, his	Developer	India				X	X	X			X	X
D3	he, him, his	Developer	India				X		X			X	X
D4	he, him, his	Developer	India		X			X					
M2	he, him, his	Manager	India				X						
I1	he, him, his	Infrastructure engineer	India								X	X	

in the process) and whether the sizeable number of additional users that are planned to be added would cause the load to go higher yet or would not impact this part of the system.

As other examples, the team spent time discussing an architectural issue in which unusually high traffic from one client could render other clients unable to use the system, reviewing a new feature being proposed by one of the developers, considering how to reduce the number of idle testing environments that were still incurring cost for the company, reflecting on the cause of an upgrade failure, and refining a ticket that had been a placeholder for the team needing to develop a permanent solution for a CPU utilization issue. This variety is indicative of the unique role that the weekly maintenance meetings serve: the issues are clearly distinct from what one may find in a typical stand up or sprint planning meeting, represent important issues in the day-to-day operations of the project, require deliberate and thoughtful conversation, yet are often not large enough to warrant a meeting of their own.

3 METHODOLOGY

Our study is a single exploratory case study [28] that follows a constructivist approach. Instead of verifying previously established theory, our study centers on exploring and understanding a particular phenomenon in its natural setting [28]. All ten WebEx videos were transcribed by a professional transcription service and we used the transcriptions and the videos as the sole sources for our analyses. Our Institutional Review Board approved the study.

To answer our first research question, we performed an inductive thematic analysis [29, 39] following the guidelines stated by Cruzes and Dyba [23]. We examined the transcripts for when meeting participants verbally introduced some information into the discussion, following the Merriam-Webster definition of information: (1) knowledge obtained from investigation, study, or instruction; (2) intelligence, news; (3) facts, data. Two researchers independently performed open coding on the first meeting, after which they compared and discussed their findings to develop a first coding scheme organizing the categories of information shared. A third researcher reviewed and gave feedback on the coding scheme and the assigned codes, which led to further refinements. This process was repeated meeting-by-meeting, leading to incremental refinements to the coding scheme. Any changes to the coding scheme led to re-coding of prior meetings to reflect the changes that were made. Throughout, the two researchers used a process of negotiated agreement [37] for their independent coding. When they could not reach an agreement, the third researcher was consulted.

Once all ten meetings were fully coded, two researchers worked together to perform axial coding, examining the internal consistency of each category of information as well as potential overlaps among categories. A few categories were merged and several assigned codes were changed to be consistent with one another.

To answer the second research question, we analyzed the discussion before a piece of information was mentioned to identify whether the team member shared it voluntarily or in response to a request from another participant. We also identified whether the information being shared was visible on the shared screen in WebEx and thus presumably referenced, or was brought into the meeting by other means. Finally, we analyzed the discussion immediately after some information was shared, because on a few occasions the original answer was corrected by another participant. Because little ambiguity exists in making these determinations, one researcher performed this analysis with another verifying the results.

To answer the third research question, we followed a process similar to the first research question, involving all three researchers in a similarly iterative process of incremental inductive thematic coding and review. This time, we sought to determine the various ways in which the team concluded the discussion of each topic (e.g., it completed the discussion with nothing further needed; after discussing, it delegated work to someone not attending the meeting or to someone who did attend and volunteered to take care of the task; after discussing for a while, it deferred the topic).

Finally, to answer the fourth research question, we analyzed the content of the WebEx videos leading up to and immediately after each discussion concluded to assess whether the team documented aspects of its deliberations and decisions. We examined the content of the screen being shared to identify what kinds of notes and/or tool actions the participant sharing the screen took publicly.

Altogether, the participants engaged in more than 3750 conversational turns (switches in speaker) to which our analyses assigned over 6500 codes. All coding was performed in MAXQDA [44]. For confidentiality reasons, the healthcare company that provided the data does not allow us to share the videos or transcripts that we analyzed. We do, however, have permission to share the anonymized extracts from the transcripts and anonymized screenshots of the

videos included in this paper. Note that some aspects of the screenshots are blurry to obfuscate sensitive information. The resulting coding schemes and expanded versions of the anonymized extracts are available as auxiliary materials along with the paper, at the following URL: https://doi.org/10.5061/dryad.9w0vt4bn8. To perform member checking, a near-final version of the paper was shared with one of the architects and the CTO who instituted the architecture committee. Each was asked to read the paper carefully and to focus on whether our description of the meetings and discussions was accurate.

4 RESULTS

In this section, we present the results of our analyses. The section is organized along the four research questions stated in Section 1.

4.1 What Kinds of Information?

Table 2 presents the 36 categories of information that we identified as being relied upon by participants in the meetings. Team members verbally introduced 694 distinct pieces of information across the meetings, meaning that on average at least once a minute some information was shared by someone. The types of information represent a wide range. Types include information pertaining to system execution (e.g., Deployment fact, Run-time fact), the state of development (e.g., Feature request, Development progress), the code itself (e.g., Architecture fact, Code fact), the development process (e.g., Team process, Testing management), clients (e.g., Customer cost, Customer context), and more (e.g., Product metadata, Internal costs).

A detailed description of each kind of information is provided in the auxiliary materials associated with this paper. Here, we highlight two particularly subtle differences. First, whereas facts concern information that is objectively "true" and thus can be verified by looking something up, assessments concern information that is more subjective, but nonetheless verifiable by investigating oneself and drawing one's own conclusion based on the investigation. An example Deployment fact is the following (we use underlining to indicate what we coded as information):

> **A1**: *Um, well their costs won't change because, um, we haven't changed the size of their database.*

whereas an example Deployment quality assessment highlights the more subjective nature of assessments:

> **A1**: *Our, um, Postgres tuning is already pretty bad.*

In this case, one can go examine the facts of what kind of tuning is in place, but then still needs to interpret what they see as to whether it is indeed pretty bad.

The second subtle difference is between facts and management. Compared to facts, which concern the state of something, e.g., the code, a test case, or the deployed software, management concerns information regarding prior actions that were taken or that should be taken given certain situations. An example of Deployment management, then, is the following:

> **M1**: *Hey, A2. I'm just curious how did the Redis get reset?*
> **A2**: *Well, do you want to explain the – the – the background?*
> **D2**: *So, actually whenever we do any access or additional ID setup, we have to, um, clear the cache, Redis cache.*

In response to a question, the developer explains that clearing the REDIS cache is a necessary step after they do any kind of work that relates to access and user IDs, because otherwise the cache holds old data.

The kinds of information shared most often reflect the software being in use (Run-time fact, Deployment management, Deployment fact, Issue detail) and under active development (Development progress, Code fact). The team performs its work by accounting for what is happening at their customers, considering what kind of effect decisions might have on the customers' use, and addressing emerging issues in the field. The product's owner, for instance, brought up a key point as the team was debating whether and how to scale some cloud service (Customer context):

> **O1**: *[client name] reached out to me today, and in the next few months they were thinking of onboarding a few more, um, of their clients, which would potentially double the number of calls.*

A bit later this led to one of the architects reminding the team that the current load on one of the servers involved was near its maximum already (Run-time fact):

> **A1**: *What if they – what if they increase the size of their database? We're already heading 90% during...*

In this context, too, we often observed the team sharing facts about the current state of the code when they deliberated how to tackle certain problems. The following Code fact illustrates:

> **D1**: *So, essentially, that means that I'm going back to the API, [component name] API, and it is responding back to me with a list of applications. So, that is one response time.*

In this case, the developer walks through how the current code works to discuss where they may be able to make changes.

Some types of information have been advocated by the design rationale literature as important to capture for later (e.g., decisions [8, 38], alternatives [10, 88], rationale [16, 57]). Besides 17 instances of Argument, which represents an underlying reason for some past action, we did not witness these kinds of information being brought back. Instead of referring to the actual decision or constraint that was made, the participants refer to the current state of the code that embeds that decision or constraint. That is, the team relies on what in many ways are the manifestations of past deliberations. The following Architectural fact, for instance, is clearly the result of an important decision made in the past:

> **A1**: *So, because both, um, you know, tenants or clients, whatever, share the same compute layer, um, it is possible for one client to, uh, negatively affect – affect the other...*

The original decision, which concerned a choice of architectural style and associated cloud-based infrastructure, shows through, but it itself is not being recounted here.

Some topics relied on a large amount of information, with the top five topics involving 52, 51, 48, 48, and 48 times, respectively, that some information was brought up by a team member. Topics that involved devising a solution to a problem, performing a postmortem, and discussions about automating certain activities in the development and deployment process involved on average the most information being shared in the discussion. Topics that involved planning how to triage tickets, defining a future meeting agenda, and sharing information about future projects involved the least

Table 2: Frequency of Different Kinds of Information Verbally Introduced (694 Total).

Category	#	%	Category	#	%	Category	#	%
Run-time fact	66	9.5%	Issue	21	3.0%	Misinformation	8	1.2%
Development progress	53	7.6%	Argument	17	2.4%	Analogous solution	7	1.0%
Code fact	48	6.9%	Testing progress	17	2.4%	Documentation progress	6	0.9%
Deployment management	47	6.8%	Change difficulty	16	2.3%	Customer cost	5	0.7%
Deployment fact	43	6.2%	Prior issue	13	1.9%	Functionality request	5	0.7%
Issue detail	43	6.2%	Deployment qlty assessment	12	1.7%	Code quality assessment	4	0.6%
Customer context	39	5.6%	Testing management	12	1.7%	Documentation qlty assmnt.	4	0.6%
Infrastructure funct.	36	5.2%	General programming knwl.	11	1.6%	Architectural qlty assmnt.	3	0.4%
Team housekeeping	35	5.0%	Internal cost	11	1.6%	Product metadata	2	0.3%
Testing fact	27	3.9%	People expertise	11	1.6%	External dev. progress	1	0.1%
Architectural fact	25	3.6%	Best practice	10	1.4%	Infrastructure progress	1	0.1%
Team process	24	3.5%	Testing quality assessment	10	1.4%	Non-functional reqmnt.	1	0.1%

amount of sharing. The fact that sharing information about future projects involved among the least amount of information sharing might seem counter intuitive, but can be explained because these topics involved quick heads-ups rather than elaborate discussions.

A small but interesting category is Misinformation, which represents when someone shared some information that subsequently was corrected. This happened only eight times, but reflected pivotal moments in the discussions. Consider the following extract:

> **A1**: *I will say that, um, [component name] did not have a UI for the lab or the mappings, so. It would be something new, I guess.*
> **O1**: *So, would the –*
> **O2**: *[component name] – [component name] has it. I don't know if we are talking the same, but, um, I don't know if you're talking about these mappings. Are you talking about this?*
> **A1**: *Yeah, I can show my screen really quick.*
> **O2**: *Yeah, and even – even this mapping is there.*

While devising a solution, one of the architects asserts that some part of the system does not have a user interface. O2 corrects the architect, points out that it does, and the architect then corrects themselves and shares their screen to show that, indeed, that part of the system does have a user interface, with O2 subsequently pointing out an important aspect of the interface (the mapping). Had O2 (the shadow product owner) not brought up that the user interface exists, the team might have gone down a design path that would be superfluous.

4.2 How Is It Brought Into the Meetings?

Not all information is shared in the same way. We identified whether information was brought up voluntarily or by request, examined if each request for information was answered or left the discussion with missing information, whether any information that was shared was subsequently corrected and thus was misinformation, and if the information was shared via some tool on screen or was shared otherwise (e.g., from memory or from a tool whose content was not shared at that time).

Many times information was casually included as part of a broader point being made. Consider the following excerpt referring to an Analogous solution:

> **A1**: *We – we could do some sort of round robin. And then, if we do have an endpoint that is misbehaving, start applying back pressure, or – or exponential back-off where we – kind of like we do with the Elasticsearch. If – if you take longer than five seconds, then we're only gonna send you one message every five seconds then. Right? Something like that.*

Note how the information—in this case a reference to a similar kind of solution used elsewhere in the code—is brought up. It acts as an invitation to follow up on the reference if needed (e.g., "What did we do again?", "Explain that to me"), but in the absence of such requests it is assumed that the reference is understood and helps clarify the solution that the team member is proposing here.

Other times, information is shared to set the stage for the following discussion. Consider this example, which involves three pieces of information being shared (Issue, Run-time fact, and Functionality request):

> **I1**: *Um, so, the thing is this morning some of the environments were down. The machine processing was at scale and then could not could not serve the request in time. The [other internal team name] wanted to have a dashboard or something to detect these kind of problems.*

The infrastructure engineer brings up a problem for the team to consider, shares a fact about the state of the run-time environment, and explains that an internal team wants to be able to monitor and detect if this problem arises again. A discussion follows during which they realize they might be able to leverage a newly implemented run-time data collection tool, briefly touch upon the information that should go on the dashboard, discuss details of a possible implementation, and decide when they will work on it.

For all excerpts that we have shown thus far, the information was voluntarily contributed, that is, the team member shared it out of their own volition without any prompt by another team member. They simply included the information in the course of making a contribution to the discussion. Such voluntary information sharing was dominant (595 out of 694 total shared pieces of information were shared voluntarily). The remainder was shared by request: before someone shared the information with the team, someone in the meeting asked for it. On most occasions, the request was an explicit question, but on several occasions it was more implicit, as in this request for a Deployment fact:

> **A2**: *If you try to do 250 con – concurrent requests, you're gonna get 429-ed because we've got every endpoint limits any site from*

making, was it, 100 concurrent requests, right, 100 or 150 is the default. A hundred or 150 –
Q1*: I think it's a – I think it's 100. That's ss – good.*
A2*: – 100 concurrent requests for any site on any end-point.*

The architect never explicitly asked the team, instead they just mentioned two potential values as part of their narrative. The quality engineer felt compelled to interject and answered with the actual value. The architect did not skip a beat, continuing their train of thought with the clarified limit.

In total, 146 requests for information were made out of which 99 were answered. The remaining 47 requests, however, went unanswered (coded as Missing information to indicate that the information requested never was provided), with the discussion simply continuing. An example is the following:

A1*: Is there an NFR for how long it should take? Because I think right now, it'd probably take uh, at least a maybe like a day to process all the results and set the new type. Or should we consider doing something with the – at query time for Elasticsearch?*
O2*: Mm-hmm.*
A1*: Um, is – is there NFR for this, or is it okay for it take – maybe that will help make the decision. Um, 'cause currently, we do set the value in Elasticsearch and Postgres. But we could probably change that to do something at query time depending on what the NFR is.*

The request was for a Non-functional requirement and went unanswered. Indeed, the architect asked twice in the above fragment (which we coded only once, since we did not code repetitions) and later asked again. The team extensively deliberated what a potentially good limit might be based on a few analogous situations, but never resolved whether an NFR existed.

Missing information did not follow any particular patterns. Out of the 36 kinds of information (Table 2), for 22 of them instances exist where that kind of information was requested and not provided during the meeting. The maximum number of times some kind of information went unanswered was seven (Code fact) and the minimum one (various, including architectural fact, Internal cost, and Argument).

On a few occasions (19 out of 694) the information being shared by one participant actually is an explicit recounting of what someone else had said in another meeting or forum. As one example, one of the topics concerned clarifying a misunderstanding between two developers who were discussing an aspect of the code on Slack (a case of Misinformation, corrected by an Architectural fact):

A2*: I mean, the only – the only thing I'd – the only thing that gives me hesitation is [person 1] wrote in bold [component name] will not connect to public internet. What [person 2] said is, it will be secure over SSL which is connected to the public internet.*

In all five examples in this section thus far, the information was shared from memory or from a source not shared on screen (in the Slack example, the architect recounted what they had seen prior in the day). This was the dominant case: out of all 694 pieces of information that were brought into the meetings, only 75 were explicitly visible in the content of the tool that was at that time shared on the WebEx screen. Of those 75, sometimes a team member would explicitly reference the content that was visible or even actively use the tool to navigate to the desired information, as in the example in Section 2 involving AWS. Other times, the information was visible on the screen but conversation proceeded without evidence that the team used or referenced the tool and its content.

Team members used a variety of tools, including Jira (which was used to share 17 pieces of information), Confluence (17), E-mail/chat (15), AWS tools (11), deployment/monitoring tools (11), their office suite (1), and proprietary tools they had developed themselves (3). A typical use of the tools was to set the stage for a topic discussion. The excerpt below provides an example (Issue):

A1*: So, I think that's worth talking about. Um, basically, like 12 days ago, um, [person name] was trying to send, uh, automated workflows, uh, out of the [client name 1] system, but the entire system was being, uh, clogged by [client name 2] or, you know, another phrase for it is like a noisy neighbor.*

In this case, a team member shares the background of a reported issue to kick off a root cause analysis by summarizing the content from a Confluence page where one developer raised the issue and others had already contributed notes documenting the issue and its undesirable behavior.

While Jira, as an issue tracking platform, is ideally suited for this kind of use, it is interesting to observe that the team had set up a set of pages in Confluence that it calls "Ask an Architect" (shown in Figure 1). These pages were designed as an explicit channel for anyone in the team or even beyond the team to directly bring issues to the architecture committee, whether it concerns something one is not sure about, asking for design help, or verifying assumptions one might have about the code. This avoids issues becoming lost in the much larger set of issues that Jira tracks and also provides an informal way to reach out to the team. The architects check these pages regularly and anybody can add notes. In the case of Figure 1, the issue was newly reported on March 12, 2020, with a description in the fourth column. Notes have already been added in the fifth column, giving the team some starting points for the discussion.

Sometimes, the team used tools other than Jira of Confluence to introduce a topic to be discussed. On one occasion, for instance, the meeting participants brought up Slack and showed an ongoing discussion in a Slack channel that was indicative of an emerging issue in the field that was not yet recorded in Jira or Confluence.

Another use of tools was to provide illustrations that helped the team understand the behavior of the deployed system. They used either standard AWS tooling to gain insight into the resource use of their cloud application or would bring up a monitoring tool they had connected to their own logging infrastructure for detailed insight into code-level behavior. As an example, they studied test results to remind themselves of what the various parts of the test suite covered (four Testing facts: the first is shared spontaneously, the second a request, the third an answer, the fourth expanded detail):

A1*: So, here are the two test runs, [A2], and it looks like DB or sorry, Merge runs everything for UI.*
A2*: Oh, okay. We got live site there?*
A1*: Yeah.*
A2*: Live site, local code, local test. Okay. It's just – it's just everything except all the, uh, mm, perfect.*

On seven occasions, the team brought up a deployment or monitoring tool specifically in response to a request being made, with the team subsequently using the tool to find the answer to the request.

12 Mar 2020	NEW	[confidential]	On 3/12 [confidential] could not get its AWF to trigger for [confidential] because AWF services were busy processing [confidential]. I just wanted to bring to your notice so that you could discuss about this and plan to introduce a minimum level of QoS for each environment.	• Currently one queue handles both evaluation and action • Actions should be queued • External system changes

Figure 1: Recreation of a Video Capture of a Snippet from Ask an Architect in Confluence.

4.3 What Discussion Outcomes?

Not all topic discussions conclude in the same manner. To anchor the analysis of our final research question—what is captured in maintenance meetings—we therefore first studied the various outcomes at which the team eventually arrives for each of the topics.

We identified four different outcomes: topics that were RESOLVED, where either what needs to be done next is fully understood and written down in a new ticket or updated in an existing ticket for downstream work by a programmer, or the question that was posed to the team is fully answered online (e.g., in Ask an Architect) or in person; DELEGATED, where the outcome as to what to do next has become clear, the team could keep working together to fully resolve the topic, but someone volunteers or is assigned to do the work of creating tickets, writing up an answer, etc. outside of the meeting; DEFERRED, where the team does not have sufficient information to determine a satisfactory path forward and tasks someone with gathering additional information to determine whether the issue is a non-issue or needs to be further discussed by the team; and UPDATE COMPLETE, when someone finished giving an update to the team, possibly answering some questions in the process. In a few cases we were unable to determine the outcome UNKNOWN. Table 3 shows the number of each resolution type per all fourteen discussion types identified in our prior work [4]. We discuss examples of each in the below, except for UNKNOWN for obvious reasons.

Table 3: Outcome per Type of Discussion.

Discussion Type	Resolved	Delegated	Deferred	Update completed	Unknown	Total
Assess problem		1	3		1	5
Automate activities	2					2
Clarify misunderstanding	1	1				2
Define internal practices	1	1		1	1	4
Devise solution	1	1				2
Gather knowledge			1		1	2
Manage accounts		3				3
Manage computatnl. resources	1					1
Perform post-mortem	2			1		3
Plan future meeting agenda	1					1
Plan how to triage tickets		1				1
Refine ticket	11				2	13
Review design proposal	2	1			1	4
Share info. about future projects				2		2
Totals per type	22	9	4	4	6	45

Approximately half the topics that the team handled over the ten meetings were resolved, meaning that it considered the discussion finished, with an answer provided or a clear action item to be worked on downstream decided upon. Resolved issues sometimes involved someone typing an answer to a question (e.g., in Ask an Architect) and changing the status of the question from "new" to "done". At other times, the team added a new ticket to Jira or updated an existing ticket. These tickets were not necessarily assigned to someone during the meeting, but the team felt that the issue was now sufficiently documented in the ticket so that, during a next round of sprint planning, the ticket can be properly considered and scheduled for implementation.

Sometimes someone on the team would resolve the issue during the meeting. As one example, the team observed they should remove running instances of their testing environment that they no longer needed, but were still using up resources and thus incurring cost to the company because their testing environments (and deployed environments) run in an external cloud service. In this case, a team member who was not sharing their screen stated that they had performed the removal on the spot:

A2: *Right. [software name] is gone, it is terminated.*
A1: *Thank you.*

Note that half the resolved cases concerned refinement of tickets. The ninth meeting was completely dedicated to considering a suite of tickets in Jira that had come in, were incomplete or not fully understood yet, and needed to be discussed by the team in order to figure out what was going on and what to do with each ticket. In a few cases, the discussion revealed that the new but incomplete ticket described an issue that the team currently was working on or had already completed under another ticket; these tickets were closed. In other cases, the tickets led the team to perform investigative work or discuss amongst themselves what the ticket might be about, followed by the team making updates to the tickets. Various other types of discussions were resolved as well, including a design proposal review that completed, two post-mortems, and a change to an internal practice upon which they agreed in the meeting.

Delegated issues involved situations where the discussion in the meeting did not fully complete, but was completed sufficiently so that the team as a whole did not need to continue deliberating at length. In other words, the team would reach a point where a broad consensus was reached about the way forward, but not all aspects were considered yet or some details needed to be looked at before the outcome could be finalized. In such cases, the team asked someone—or someone volunteered—to take it forward on their own. In the following example, the product owner volunteered to create a ticket based on the discussion the team had just had, recognizing that they need to get a little bit of additional feedback from elsewhere before doing so:

O1: *Okay. Um, I think this is good. [person name 1] is meeting with [person name 2] to go over it tomorrow. Um, we'll get some*

feedback from here, and I guess we'll just go from there. But, um, in the meantime, I'll create this ticket to bring to hmm, actually yeah, I'll just create this ticket.

On a few occasions, delegation also involved the team deciding that a certain issue was not theirs to fix, but belonged to another team. Overall, delegation occurred in nine out of 45 cases, meaning that together with the number of resolved cases, the team successfully addressed two-thirds of the topics that came its way.

As compared to delegation, when the team's discussion leads to it identifying the way forward, deferred issues are where the discussion is inconclusive; this happened on four occasions. Typically, the team felt that it needed additional information that would help it decide what they were facing. Consider the following example:

A1*: So, yeah, I think the more load is fine, probably.*
A2*: Oh, yeah. Again – Yeah, if they double it – I don't – I don't think we'll – We'll just have to monitor and make sure it's okay.*

The topic concerns a client planning to add a significant number of new users, with the team concerned that it may cause some of their servers to be unable to handle the additional load that the new users would incur. The underlying issue is that, if the load becomes too high, it would require a re-architecting of some of their software. The discussion, though extensive, cannot predict what the future server load will likely be, though they feel it should probably be okay. As a result, they defer the issue, deciding to monitor the server instead, and only planning to return to the discussion if the load indeed becomes problematic.

We examined whether missing information (verbal requests for information that were not answered in the meeting, see Section 4.2) might have played a role in some topics being deferred. Across the four deferred topics, only four requests for information went unanswered, which is right at the average of one unanswered request per topic, indicating that it is unlikely that it causes deferral. Indeed, upon close inspection, the four deferrals all concerned situations where the necessary information was not available at the time and needed to be collected in future, as in the example above.

Updates took place four times and represented short briefings where someone updates the team on an issue they had been working on or a future plan. The following excerpt presents an example of how an update on the state of application security was started by one of the architects.

A1*: Um, cool. Um, AppSec update, I just got this today. [person name] is always very bad at planning ahead, I guess. I have a meeting tomorrow at 10:00 to go over next steps. Um, it looks like be doing planning, essentially.*

4.4 Captured for Future Reference?

A large variety of information was captured for future reference in support of the outcomes upon which the team decided. In total, the team captured 186 pieces of information of 17 different kinds (Table 4). The most frequent was Idea/alternative, with an example provided in Figure 2. In this case, the team was discussing the issue shown in Figure 1 and captured aspects of the discussion by editing the fifth column in Ask an Architect for this issue. Two ideas were raised in the discussion. The first was to attempt auto retry (about two-thirds down in the notes), the second to use one queue per tenant (near the bottom). Both ideas were captured along with a variety of other things, such as a Code state capturing how the code currently works ("Currently one queue handles both evaluation and action") and a Problem background/status of which the architects reminded the team ("Actions should be queued"), each listed at the top of Figure 2. One of the architects took these notes throughout the discussion, with the notes visible on the screen being shared over WebEx so all team members could see. Eventually, the team preferred the first idea and the second was crossed out. Eventually, too, the team felt the issue was satisfactory resolved and changed the status of the item from "NEW" (column 2 in Figure 1) to "DONE" (Metadata). The team also linked (Metadata) the item in Ask an Architect to a Jira ticket it created in the meeting to capture the decision to create a proof-of-concept splitting the evaluator and action services (Issue/ticket high-level description).

Ask an Architect was not the only way in which the team used Confluence to document important information for future reference. The team also used what it calls *playbook entries* to record system documentation, *notes pages* to hold meeting notes, and *topic pages* to document specialized knowledge concerning important subjects. Interestingly, despite the goal of capturing notes for every meeting and despite the template for notes pages, they kept formal notes in only two meetings; all other times, no notes pages were created, although information was captured in other ways as the above example of using Ask an Architect illustrates. One of the two cases where they did create a notes page concerned a post-mortem, which had the explicit goal of documenting ways in which the team could improve aspects of the process it uses to address situations in which someone breaks the master build. The debrief resulted in a new process (Plan of action) that they detailed, including which leaders should be notified if the problem recurs (Best practice).

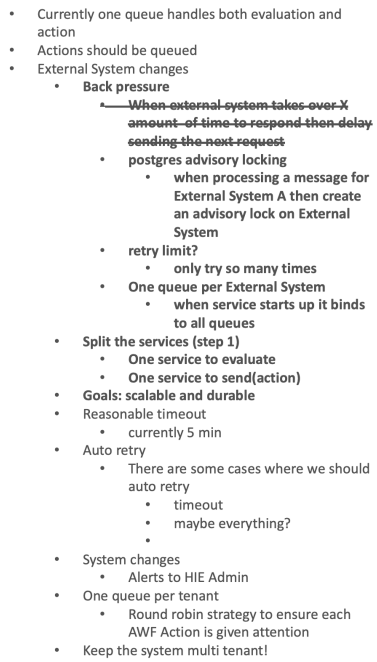

Figure 2: Recreation of a Video Capture of Code Fact and Two Ideas/alternatives Being Documented in Ask an Architect.

Table 4: Frequency of Different Kinds of Information Captured (186 Total).

Category	#	%	Category	#	%
Idea/alternative	36	19.4%	Action item	7	3.8%
Team process	23	12.4%	Issue/ticket high-level description	5	2.7%
Metadata	19	10.2%	Implementation goal/scope	4	2.2%
Discussion item	19	10.2%	Scheduling estimate	3	1.6%
Design feedback	18	9.7%	Requirement	3	1.6%
Code state	14	7.5%	Impact estimate	2	1.1%
Implementation roadmap	11	5.9%	Administrative decision	2	1.1%
Rationale	9	4.8%	Best practice	2	1.1%
Situation's background/status	9	4.8%			

Compared to Confluence, which was used to capture 141 pieces of information, Jira was used 44 times. Given its role as a repository of issues to be addressed in the code base, this is not too surprising, because not everything the team discusses directly results in specific code tasks. Still, the kind of information captured in Jira was broader than just Issue/high-level description, with Idea/alternative, Design feedback, Rationale, and Implementation goal/scope also happening six, six, six, and three times, respectively.

Figure 3 provides a final example of how the team works and documents its outcomes. As part of a discussion to improve the test suite of the system, the team created a topic page in Confluence and took notes as they settled on the overall Implementation roadmap (only a small part shown) and Scheduling estimate. They included Rationale as to why they should do a proof-of-concept now, not later, and left as an Action item that someone should take the roadmap and turn it into specific Jira tickets.

While a great deal of information is captured, it is not done for every topic. Capture took place for 26 out of 45 topics. A few interesting patterns exist in when the team does and does not document. When work was delegated, for instance, discussion information was only captured a third of the time and never captured to whom the topic was delegated; it was assumed that someone would remember and inform the person. For deferred topics, no information was recorded at all; the same was true for topics that were updates to the team.

The type of discussion held also seemed to influence whether information was captured. Surprisingly, none of the topics focusing on assessing a problem (five) or gathering knowledge (two) involved information capture. Both of these types of discussions are focused on obtaining information that the team does not have, which should make it prudent for the team to document what it learns in the process so it can be used at a later time. It seems that in these cases, however, the team is content with discussing the topic and letting whomever led them take the information forward.

5 DISCUSSION

Our study is novel in focusing on software maintenance meetings, a type of meeting that to date has not been studied in depth. Similar to Ko et al. [42], who catalogued the types of information developers seek, we chose to focus on the role of information. Ko et al. found 21 types of information sought, with about one third focused on code. Our study reveals an even broader set of information relevant to software maintenance meetings, but with much less focus on code and more of a focus on run-time and deployment information as well as architecture and process-related information. This is not surprising, given the higher-level tasks in which the architecture committee engaged and given that the team works on a cloud-based system. Our study further differs from Ko et al. in also documenting the resolution of the topic discussions as well as the information capture practices in which the team engaged.

Recurring meetings [60] are a regular in software organizations with, for instance, Agile stand-ups [80, 81], bug triage meetings [78], and weekly status meetings [47] all having been studied before. The software maintenance meetings we examined sometimes involved activities somewhat similar to what takes place in these other kinds of meetings (e.g., a progress update for a fix for a client, some early pre-triaging). On the whole, however, the meetings exhibited a strong focus on planning and problem-solving topics surrounding the deployment, operation, and evolution of the system, instead of updates from individuals as to how their tasks are coming along. Our study, then, uniquely documents the effect of this different focus on the nature and breadth of information being shared.

The software maintenance meetings we studied take place in the context of a variety of other meetings at the organization (see Section 2). This is in line with Lavalee and Robillard [47], who observed a rich interconnected set of meetings taking place in their study and argued that it is important to build an understanding of the information flow among these meetings. Our result offer a basis for creating this understanding by documenting precisely what information flows in and out of software maintenance meetings, which serve an essential role in the overall ecosystem of meetings.

In the remainder of this section, we first recap the main findings and then discuss the implications of our study.

Takeaway 1: *The deliberations taking place in maintenance meetings rely on a large amount of highly varied information.* On average, some piece of information was brought up approximately once a minute, with the diversity of the topics being addressed shining through in the diversity of the kinds of information shared. The fact that the system is already deployed leads to much information being 'fleeting' in nature, representing the current state of deployment or code base. Topics that involved devising a solution to a problem, performing a post-mortem, and automating certain activities in the development and deployment process involved on average the most information being shared in the discussion.

Takeaway 2: *A significant amount of information is captured to document the outcomes of topic discussions in maintenance meetings.* On average once every four minutes some kind of information is

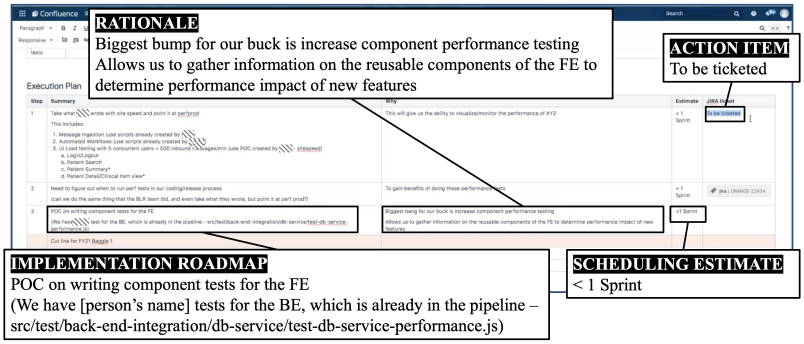

Figure 3: Information Captured during the Discussion Centered on the Design of Major Improvements to the Test Suite.

recorded. As with information sharing, the kinds of information being captured by the team vary strongly and especially vary with the type of discussion held. A significant portion pertains to design and code level guidance and decisions, but some portion also covers the team itself and how it operates and addresses emergent issues.

Takeaway 3: *Yet, only 60% of the topics that were discussed involved information capture for later reference.* Even though there was a process and standard format for taking notes, the prescribed Confluence notes pages were only used in two meetings. Only for one-third of the topics involving delegation was anything captured and nothing was captured for the topics that were deferred.

Takeaway 4: *People are crucial to managing the meetings' information.* With only about 10% of information being shared visible in a tool on the shared screen, individual participants play a key role in sharing information either from memory or from another tool or source they have locally open on their computer. Most requests for information were similarly answered by a participant without referencing the shared screen. And, with 40% of the topics not leading to any notes being taken, much of the responsibility in taking discussion outcomes forward falls on the individual team members.

Takeaway 5: *Tools nonetheless play an important role.* Confluence and Jira were the dominant tools for capturing information, documenting outcomes and important aspects of the discussions. Confluence and Jira also served an important role in agenda setting for the topics being discussed, though other tools were also used to locate and bring relevant information into the meetings, with AWS and several home-grown tools essential to understanding the actual behavior of the deployed system and associated code base.

5.1 Implications for Practice

As stated earlier, the team we studied is considered high-performing by its leadership. We see this echoed in the outcomes: two-thirds of the topics are fully resolved or delegated for final resolution, and the deferred topics are not borne out of problems in the discussion but represent the genuine case of needing to acquire future information by monitoring the system. The few remaining topics were either updates (no resolution necessary) or we could not deduce the resolution from the data. In this context, we discuss what our findings might mean for other teams involved in maintenance meetings.

Practice 1: *Ensure the right team members are present.* It is essential that maintenance meetings involve knowledgeable personnel. In the case of the team we studied, the two architects shared a majority of the information (75%), partly because how long they had been with the team. Other team members, however, were explicitly invited to the meeting for their specialized knowledge.

Practice 2: *Use tools such as Confluence and Jira, but surround them with meaningful structure.* Both in terms of kicking off topic discussions and in capturing important outflow from these discussions, the team established practices through which not just the team but others who may need to invoke the team can easily approach it. Ask an Architect is the most powerful example, but the team also explicitly spent time on how to best triage, re-considering established practices in testing and handling emergencies at clients, and documenting team processes and best practices. Such reflective practices characterize high-performing teams [86].

Practice 3: *Allow the instant look-up of information if necessary to the discussion.* Rather than postponing a topic, the team was effective in using its tools to dynamically bring up information about the deployed system. This was essential to a number of the discussions. Being able to do so quickly and visible to everyone was a key enabler in getting several topics resolved successfully. Such 'artifact seeding' is an important part of good practice [32].

Practice 4: *Be consistent in capturing discussion outflow.* Some teams are too reliant on individuals' memory to share important outcomes and considerations from meetings. Being more principled about taking notes, perhaps through collaborative notetaking as it has shown auxiliary benefits [21, 68, 74], would be a good general practice, particularly in light of the next suggestion.

Practice 5: *Pay attention to unanswered information requests.* While the impact of the unanswered requests for information was not a focus of our analysis, it is still possible that the team could have done better had these been answered. A side benefit of conscientiously answering all requests for information is that it can forge strong team cohesion. Ignoring others, especially when they are remote, can have negative and long-lasting effects [76].

5.2 Implications for Research

Our findings give rise to a number of different research directions that we believe are worthwhile pursuing.

Future Research 1: *Other teams.* We strongly feel that our study should be replicated on other teams with other characteristics, including, among others, a lower-performing team, a team working in a different domain, and a team working with different tools. By

juxtaposing results from such studies, more can be learned about effective and less effective practices in maintenance meetings.

Future Research 2: *Conversational analysis.* A logical next step is to perform a detailed analysis of the conversations that take place, to examine in detail the impact of the information shared on the discussions taking place. Answering questions such as if the information is relevant and how it impacts the discussions and outcomes is important to identify best practices (e.g., [5, 54]).

Future Research 3: *Tools.* Design rationale tools have long been explored to capture meeting outcomes in software development (e.g., [15, 22, 48, 85]). Given that our findings show that not all topics in maintenance meetings concern design, more general meeting capture tools as explored in the CSCW and HCI literature might be more applicable (e.g., [68, 71, 91]). With new AI-driven meeting capture tools offering automated transcription and summarization (e.g., [2, 41, 89]), opportunities exist for exploring new maintenance meeting tools that offload much of the notetaking responsibility.

Future Research 4: *Meeting ecosystem.* No meeting stands alone [3] and so it is with maintenance design: as already stated, the meetings we studied are part of a rich ecosystem of many different meetings for the team and its members. As others already noted [47], it is important to study how these meetings interconnect, particularly in terms of the information flow amongst them.

6 THREATS TO VALIDITY

Our findings are subject to a variety of threats to validity. First is the issue of **representativeness**. The 36 categories of information that we witnessed being shared may not be representative of all categories shared across all meetings held by the team. Similarly, the set of 10 meetings may not be representative of all meetings by the team. By choosing a window of nearly three months we hope to have reduced this potential issue. Other issues related to representativeness (e.g., single team, high-performing team, single company, healthcare domain, cloud-based system) are an artifact of the research methodology of a single exploratory case study. Additional studies of other teams in other situations are necessary.

Second is the threat of **incompleteness**: because only a single screen was shared at the time, it is possible that team members engaged in invisible work that could influence our findings, for instance by sharing quick notes with each other or privately capturing discussion notes for later use. Additional study is necessary to understand the prevalence and potential impact of such practices.

Third are potential concerns with the study execution in terms of the **consistency** and **stability** of the analysis process, together with the **confirmability** of the results. To counter this threat, we followed established practices in inductive thematic analysis, with two researchers performing coding independently before comparing and resolving notes and a third researcher providing independent feedback at each step. Additionally, as described in Section 3, we performed member checking. Both the architect and the CTO felt that our description of the meetings was accurate, our coding seemed appropriate, and results, though including some surprises (e.g., frequency of information sharing, limited capture of information), aligned with their perceptions.

7 RELATED WORK

Software maintenance has been studied from a broad range of perspectives, including but not limited to empirical studies of developers making code changes (e.g., [13, 14]), novel tools to understand and modify code (e.g., [26, 64]), characterizing different maintenance activities (e.g., [19, 49], visualizing code (e.g., [46, 90]), and re-engineering and refactoring (e.g., [27, 35, 58, 59]). A particularly relevant thread of work has examined the information needs of software developers in their day-to-day programming (e.g., [24, 42, 72]) and created new tools for helping them deal with these information needs (e.g., [24, 43]). Our work follows this thread and contributes a first look at both information needs and information capture in maintenance meetings.

Closely related to our work are studies of Agile meetings (e.g., [80, 81]), studies of design meetings (e.g., [25, 55, 57, 61]), and studies of hybrid and remote meetings (e.g., [6, 62, 67]). Our work is inspired by these studies documenting in detail a range of phenomena that take place in different kinds of software development meetings and follows a similar methodological approach. It is unique, however, in our focus on maintenance meetings specifically and the role that information has in setting a context for the work that takes place in these meetings.

To the best of our knowledge, no studies have been performed to date detailing the role of Confluence or Jira in software development meetings, though issue tracking has been studied extensively (e.g., [7, 12]) and the use of wikis in software development has also been a subject of study (e.g., [17]). Our findings are unique in detailing the use of these tools in software maintenance meetings.

An extensive strand of research has sought to develop tools that help capture aspects of meetings, whether specifically for software development discussions through design rationale tools (e.g., [16, 22, 40, 85]) or more broadly through generic meeting capture tools (e.g., [11, 17, 69, 77]). Our work at this time does not seek to develop a tool, but has implications for the design of such tools.

8 CONCLUSIONS

Maintenance meetings are the heartbeat of a software project: they are necessary to ensure a project keeps on track, emergent issues are addressed, and new ideas are fostered and have a place to be vetted. This paper contributes a first look at the kind of information that flows into software maintenance meetings, how it flows into those meetings, what kinds of outcomes result from the discussions in them, and what information is captured in the meetings to support downstream activities. Among others, our findings include that: (1) developers rely on a wide variety of information in the discussions in these meetings, (2) they use a range of tools to share additional information, (3) they successfully address most topics brought up in the meetings, and (4) only 60% of the topics involve information being captured for later use.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants CCF-2210812 and CCF-2210813.

REFERENCES

[1] 2023. Engineering Meetings: Tips to Help Your Team | Range. https://www.range.co/blog/engineering-meeting Accessed on 07/2023.

[2] 2023. Otter.ai - Voice Meeting Notes & Real-time Transcription. https://otter.ai/ Accessed on 08/2022.

[3] Joseph A. Allen, Tammy Beck, Cliff W. Scott, and Steven G. Rogelberg. 2014. Understanding workplace meetings: A qualitative taxonomy of meeting purposes. *Management Research Review* 37, 9 (Jan. 2014), 791–814. Publisher: Emerald Group Publishing Limited.

[4] Anon Anon. 2022. removed for blind review.

[5] Alex Baker and André van der Hoek. 2010. Ideas, subjects, and cycles as lenses for understanding the software design process. *Design Studies* 31, 6 (Nov. 2010), 590–613.

[6] Jose Maria Barrero, Nicholas Bloom, and Steven J. Davis. 2021. *Why working from home will stick.* Technical Report 28731. National Bureau of Economic Research. https://www.nber.org/papers/w28731

[7] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. 2010. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work (CSCW '10)*. ACM, 291–300.

[8] Manoj Bhat, Klym Shumaiev, and Florian Matthes. 2017. Towards a Framework for Managing Architectural Design Decisions. In *11th European Conference on Software Architecture: Companion Proceedings (ECSA '17)*. 48–51.

[9] Tingting Bi, Wei Ding, Peng Liang, and Antony Tang. 2021. Architecture information communication in two OSS projects: The why, who, when, and what. *Journal of Systems and Software* 181 (Nov. 2021), 111035.

[10] Daniel G. Bobrow and Ira P. Goldstein. 1980. Representing design alternatives. In *Proceedings of the 1980 AISB Conference on Artificial Intelligence (AISB'80)*. IOS Press, 25–35.

[11] Gerald Bortis. 2010. Informal software design knowledge reuse. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. 385–388.

[12] Gerald Bortis and André van der Hoek. 2013. PorchLight: A tag-based approach to bug triaging. In *2013 35th International Conference on Software Engineering (ICSE)*. 342–351.

[13] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the characteristics of vulnerable code changes: an empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 257–268.

[14] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, 1589–1598.

[15] Janet Burge and David Brown. 2008. SEURAT. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 835–838.

[16] Janet E. Burge, John M. Carroll, Raymond McCall, and Ivan Mistrik. 2008. *Rationale-Based Software Engineering*. Springer Berlin Heidelberg.

[17] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 2016. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software* 116 (June 2016), 191–205.

[18] Andrew Chan, Karon MacLean, and Joanna McGrenere. 2008. Designing haptic icons to support collaborative turn-taking. *International Journal of Human-Computer Studies* 66, 5 (2008), 333–355. Publisher: Elsevier.

[19] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. 2001. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 13, 1 (2001), 3–30.

[20] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. 2007. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings. In *SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. 557–566.

[21] Patrick Chiu, John Boreczky, Andreas Girgensohn, and Don Kimber. 2001. LiteMinutes: an Internet-based system for multimedia meeting minutes. In *Proceedings of the 10th international conference on World Wide Web*. 140–149.

[22] E. Jeffrey Conklin and K. C. Burgess Yakemovic. 1991. A process-oriented approach to design rationale. *Human-Computer Interaction* 6, 3 (Sept. 1991), 357–391.

[23] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 275–284.

[24] Brian De Alwis. 2008. Supporting conceptual queries over integrated sources of program information. (2008). Publisher: University of British Columbia.

[25] Uri Dekel. 2005. Supporting distributed software design meetings: what can we learn from co-located meetings? *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–7.

[26] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 341–350.

[27] B. Du Bois, S. Demeyer, and J. Verelst. 2004. Refactoring - improving coupling and cohesion of existing code. In *11th Working Conference on Reverse Engineering*. IEEE Comput. Soc, 144–151.

[28] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer, 285–311.

[29] Jennifer Fereday and Eimear Muir-Cochrane. 2006. Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development. *International Journal of Qualitative Methods* 5, 1 (March 2006), 80–92. Publisher: SAGE Publications Inc.

[30] Mark Ferlatte. 2017. Well Met: the Software Engineering Meetings You Actually Need — Truss. https://truss.works/blog/2017/2/3/well-met-the-software-engineering-meetings-you-actually-need Accessed on 07/2023.

[31] Matthew Finnegan. 2022. For developers, too many meetings, too little 'focus' time. https://www.computerworld.com/article/3669911/for-developers-too-many-meetings-too-little-focus-time.html

[32] Gerhard Fischer, Jonathan Grudin, Raymond McCall, Jonathan Ostwald, David Redmiles, Brent Reeves, and Frank Shipman. 2001. Seeding, evolutionary growth and reseeding: The incremental development of collaborative design environments. *Coordination theory and collaboration technology* 447 (2001), 472. Publisher: Citeseer.

[33] Frederik M. Fowler. 2019. The Sprint Planning Meeting. In *Navigating Hybrid Scrum Environments*. Apress, 83–88. http://link.springer.com/10.1007/978-1-4842-4164-6_13

[34] Frederik M. Fowler. 2019. The Sprint Retrospective. In *Navigating Hybrid Scrum Environments*. Apress, 97–100. http://link.springer.com/10.1007/978-1-4842-4164-6_16

[35] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley.

[36] Thomas Fritz and Gail C. Murphy. 2010. Using information fragments to answer the questions developers ask. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 175–184.

[37] D. R. Garrison, M. Cleveland-Innes, Marguerite Koole, and James Kappelman. 2006. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education* 9, 1 (Jan. 2006), 1–8.

[38] Fabian Gilson, Sam Annand, and Jack Steel. 2020. Recording Software Design Decisions on the Fly. In *Joint Proceedings of SEED & NLPaSE*. 53–66.

[39] Greg Guest, Kathleen M. MacQueen, and Emily E. Namey. 2011. *Applied Thematic Analysis.* SAGE Publications.

[40] G. P Heliades and E. A Edmonds. 1999. On facilitating knowledge transfer in software design. *Knowledge-Based Systems* 12, 7 (1999), 391–395.

[41] Francois Jacquenet, Marc Bernard, and Christine Largeron. 2019. Meeting Summarization, A Challenge for Deep Learning. In *Advances in Computational Intelligence (Lecture Notes in Computer Science)*, Ignacio Rojas, Gonzalo Joya, and Andreu Catala (Eds.). Springer International Publishing, 644–655.

[42] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 344–353.

[43] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, 151–158.

[44] Udo Kuckartz and Stefan Rädiker. 2019. *Analyzing Qualitative Data with MAXQDA: Text, Audio, and Video.* Springer.

[45] Catherine Lai, Jean Carletta, and Steve Renals. 2013. Modelling participant affect in meetings with turn-taking features. In *Proc. Workshop of Affective Social Speech Signals*.

[46] Thomas D. LaToza and Brad A. Myers. 2011. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 117–124.

[47] Mathieu Lavallée and Pierre N. Robillard. 2015. Why Good Developers Write Bad Code: An Observational Case Study of the Impacts of Organizational Factors on Software Quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 677–687.

[48] J. Lee. 1997. Design rationale systems: understanding the issues. *IEEE Expert* 12, 3 (May 1997), 78–85.

[49] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. 1978. Characteristics of application software maintenance. *Commun. ACM* 21, 6 (June 1978), 466–471.

[50] loopinhq. 2023. Weekly Tech Meeting. https://www.loopinhq.com/meeting-templates/weekly-tech-meeting Accessed on 07/2023.

[51] Marisela Gutierrez Lopez, Kris Luyten, Davy Vanacken, and Karin Coninx. 2017. Untangling Design Meetings: Artefacts as Input and Output of Design Activities. In *Proceedings of the European Conference on Cognitive Ergonomics 2017*. ACM, 176–183.

[52] Marin Luetic. 2023. 15 topics to discuss during weekly developer team meetings. https://decode.agency/article/development-team-meeting-topics/ Accessed on 07/2023.

[53] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. 2015. How Software Designers Interact with Sketches at the Whiteboard. *IEEE Transactions on Software Engineering* 41, 2 (Feb. 2015), 135–156.

[54] Umme Ayda Mannan, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. On the relationship between design discussions and design quality: a case study of Apache projects. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, 543–555.

[55] Gloria Mark. 2002. Extreme collaboration. *Commun. ACM* 45, 6 (2002), 89–93.

[56] Stanic Mislav. 2023. 7 important software engineering meetings you actually need. https://www.shakebugs.com/blog/software-engineering-meetings/#The_technical_meeting Accessed on 07/2023.

[57] Thomas P. Moran and John M. Carroll. 2020. *Design Rationale: Concepts, Techniques, and Use.* CRC Press.

[58] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 2008. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. In *Balancing Agility and Formalism in Software Engineering (Lecture Notes in Computer Science)*, Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz Walter (Eds.). Springer, 252–266.

[59] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. 2006. Does Refactoring Improve Reusability?. In *Reuse of Off-the-Shelf Components (Lecture Notes in Computer Science)*, Maurizio Morisio (Ed.). Springer, 287–297.

[60] Karin Niemantsverdriet and Thomas Erickson. 2017. Recurring Meetings: An Experiential Account of Repeating Meetings in a Large Organization. *Proceedings of the ACM on Human-Computer Interaction* 1, CSCW (Dec. 2017), 84:1–84:17.

[61] Gary M. Olson, Judith S. Olson, Mark R. Carter, and Marianne Storrosten. 1992. Small Group Design Meetings: An Analysis of Collaboration. *Human–Computer Interaction* 7, 4 (1992), 347–374.

[62] K Parker, J Horowitz, and R Minkin. 2020. How the Coronavirus Outbreak Has – and Hasn't – Changed the Way Americans Work. https://www.pewresearch.org/social-trends/2020/12/09/how-the-coronavirus-outbreak-has-and-hasnt-changed-the-way-americans-work/ Accessed on 02/2023.

[63] V.T. Rajlich and K.H. Bennett. 2000. A staged model for the software life cycle. *Computer* 33, 7 (July 2000), 66–71.

[64] Sarah Rastkar and Gail C. Murphy. 2013. Why did this code change?. In *2013 35th International Conference on Software Engineering (ICSE)*. 1193–1196.

[65] Sean Rintel, Priscilla Wong, Advait Sarkar, and Abigail Sellen. 2020. *Methodology and Participation for 2020 Diary Study of Microsoft Employees Experiences in Remote Meetings During COVID-19.* Technical Report. 46 pages. https://www.microsoft.com/en-us/research/uploads/prod/2020/10/2020-10-FOW-SIM1-RemoteMeetingsDuringCOVID19-MethodologyAndParticipation.pdf

[66] G. Ruhe and M.O. Saliu. 2005. The Art and Science of Software Release Planning. *IEEE Software* 22, 6 (Nov. 2005), 47–53.

[67] Banu Saatçi, Kaya Akyüz, Sean Rintel, and Clemens Nylandsted Klokmose. 2020. (Re)Configuring Hybrid Meetings: Moving from User-Centered Design to Meeting-Centered Design. *Computer Supported Cooperative Work (CSCW)* 29, 6 (2020), 769–794.

[68] Samiha Samrose, Daniel McDuff, Robert Sim, Jina Suh, Kael Rowan, Javier Hernandez, Sean Rintel, Kevin Moynihan, and Mary Czerwinski. 2021. MeetingCoach: An Intelligent Dashboard for Supporting Effective & Inclusive Meetings. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–13.

[69] Bill N. Schilit, Lynn D. Wilcox, and Nitin "Nick" Sawhney. 1997. Merging the benefits of paper notebooks with the power of computers in dynomite. In *CHI '97 Extended Abstracts on Human Factors in Computing Systems (CHI EA '97)*. ACM, 22–23.

[70] Olga Semusheva. 2023. The 6 Most Important Project Development Meetings. https://steelkiwi.com/blog/6-most-important-project-development-meetings/ Accessed on 07/2023.

[71] Yang Shi, Chris Bryan, Sridatt Bhamidipati, Ying Zhao, Yaoxue Zhang, and Kwan-Liu Ma. 2018. MeetingVis: Visual Narratives to Assist in Recalling Meeting Context and Content. *IEEE Transactions on Visualization and Computer Graphics* 24, 6 (June 2018), 1918–1929.

[72] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering* 34, 4 (July 2008), 434–451.

[73] J. Singer. 1998. Practices of software maintenance. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 139–145.

[74] G. Singh, L. Denoue, and A. Das. 2004. Collaborative note taking. In *The 2nd IEEE International Workshop on Wireless and Mobile Technologies in Education, 2004.* 163–167.

[75] David Socha and Josh Tenenberg. 2013. Sketching Software in the Wild. In *35th International Conference on Software Engineering*. 1237–1240.

[76] Sabine Sonnentag. 2001. High performance and meeting participation: An observational study in software design teams. *Group Dynamics: Theory, Research, and Practice* 5 (2001), 3–18. Publisher: Educational Publishing Foundation.

[77] Lisa Stifelman, Barry Arons, and Chris Schmandt. 2001. The audio notebook: paper and pen interaction with structured speech. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '01)*. ACM, 182–189.

[78] Viktoria Stray. 2018. Planned and unplanned meetings in large-scale projects. In *Proceedings of the 19th International Conference on Agile Software Development: Companion (XP '18)*. ACM, 1–5.

[79] Viktoria Stray and Nils Brede Moe. 2020. Understanding coordination in global software engineering: A mixed-methods study on the use of meetings and Slack. *Journal of Systems and Software* 170 (Dec. 2020), 110717.

[80] Viktoria Stray, Nils Brede Moe, and Gunnar R. Bergersen. 2017. Are Daily Stand-up Meetings Valuable? A Survey of Developers in Software Teams. In *Agile Processes in Software Engineering and Extreme Programming (Lecture Notes in Business Information Processing)*, Hubert Baumeister, Horst Lichter, and Matthias Riebisch (Eds.). 274–281.

[81] Viktoria Stray, Dag I. K. Sjøberg, and Tore Dybå. 2016. The daily stand-up meeting: A grounded theory study. *Journal of Systems and Software* 114 (2016), 101–124.

[82] Viktoria Gulliksen Stray, Yngve Lindsjørn, and Dag I.K. Sjøberg. 2013. Obstacles to Efficient Daily Meetings in Agile Development Projects: A Case Study. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 95–102.

[83] Viktoria Gulliksen Stray, Nils Brede Moe, and Aybüke Aurum. 2012. Investigating Daily Team Meetings in Agile Software Projects. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. 274–281.

[84] Antony Tang, Aldeida Aleti, Janet Burge, and Hans van Vliet. 2010. What makes software design effective? *Design Studies* 31, 6 (Nov. 2010), 614–640.

[85] Antony Tang, Yan Jin, and Jun Han. 2007. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software* 80, 6 (2007), 918–934.

[86] Joost Visser, Sylvan Rigal, Gijs Wijnholds, and Zeeger Lubsen. 2016. *Building Software Teams: Ten Best Practices for Effective Software Development.* "O'Reilly Media, Inc.".

[87] J. Wu, T.C.N. Graham, and P.W. Smith. 2003. A study of collaboration in software design. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* IEEE Comput. Soc, 304–313.

[88] Lihua Xu, Scott A. Hendrickson, Eric Hettwer, Hadar Ziv, André van der Hoek, and Debra J. Richardson. 2006. Towards supporting the architecture design process through evaluation of design alternatives. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis (ROSATEA '06)*. ACM, 81–87.

[89] Xianjun Yang, Yan Li, Xinlu Zhang, Haifeng Chen, and Wei Cheng. 2023. Exploring the Limits of ChatGPT for Query or Aspect-based Text Summarization. http://arxiv.org/abs/2302.08081 arXiv:2302.08081 [cs].

[90] YoungSeok Yoon, Brad A. Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 119–126.

[91] Ying Zhang, Marshall Bern, Juan Liu, Kurt Partridge, Bo Begole, Bob Moore, Jim Reich, and Koji Kishimoto. 2010. Facilitating meetings with playful feedback. In *CHI'10 Extended Abstracts on Human Factors in Computing Systems*. 4033–4038.