Supporting Support Engineers

Esdras Kutomi
*Brigham Young University*

Follow this and additional works at: [https://scholarsarchive.byu.edu/etd](https://scholarsarchive.byu.edu/etd)

Part of the Physical Sciences and Mathematics Commons

**BYU ScholarsArchive Citation**

[https://scholarsarchive.byu.edu/etd/8431](https://scholarsarchive.byu.edu/etd/8431)

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Supporting Support Engineers

Esdras O. Kutomi

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Master of Science

Jonathan Sillito, Chair
Quinn Snell
Ryan Farrell

Department of Computer Science
Brigham Young University

Copyright © 2020 Esdras O. Kutomi
All Rights Reserved
ABSTRACT

Supporting Support Engineers

Esdras O. Kutomi
Department of Computer Science, BYU
Master of Science

The steady and uninterrupted availability of systems is essential for the mission of many companies and other organizations. This responsibility relies mostly upon support engineers, who are responsible to respond to incidents. Incident response is a unique type of task in software engineering, given it carries distinguishing characteristics like risks, pressure, incomplete information and urgency. Despite the importance of this task for many organizations, little can be found in the literature about the incident response task and model. To fill the gap, we created a theoretical foundation to foster research on incident response. We conducted an interview study, asking 12 support engineers about their experiences dealing with outages, service degradation, and other incidents that demanded an urgent response. We used our 22 collected cases to identify important concepts of incidents and their dimensions, and created an ontology of incidents and a model of the incident response. To validate the usefulness of our results, we analyzed our incidents based on our ontology and model, providing some insights related to detection of incidents, investigation and the hand over process. We also provide analytical insights related to the prevention of resource limitation incidents. Finally, we validate the usefulness of our research by proposing an improvement on monitoring tools used by support engineers.

Keywords: software engineering, incident response, troubleshooting, investigation
ACKNOWLEDGMENTS

Thanks to my wife Carolina Kutomi, my advisor Jonathan Sillito, my mother Mary and my family for supporting me.
Table of Contents

List of Figures vi

List of Tables vii

1 Introduction 1
   1.1 Definitions ................................................................. 2

2 Related work and background 4
   2.1 Exploratory studies in heavy cognitive tasks ....................... 4
   2.2 Studies on troubleshooting and information gathering and causes of problems 5
   2.3 Studies on tools to help support engineers .......................... 6
   2.4 Studies of training as an approach to help support engineers ....... 7
   2.5 Grounded Theory ............................................................. 8

3 Thesis Statement 10

4 Methodology 11
   4.1 Data Collection ............................................................. 11
   4.2 Data Analysis ............................................................... 14

5 Study Findings 19
   5.1 Incidents ................................................................. 19
      5.1.1 Symptoms and starting point ....................................... 19
      5.1.2 Root cause and mitigable point ................................... 21
List of Figures

5.1 How P9.I1 happened by linking events and behaviors. ......................... 22
5.2 Incident response model. ................................................................. 27
5.3 Differences of early, on time and late detection. The red line is a threshold
    representing an impact. ............................................................... 29
6.1 A common dashboard tool. ............................................................... 50
6.2 Improvement of a dashboard tool to provide information correlated to an
    incident. .................................................................................. 51
List of Tables

4.1 Attributes of incidents ........................................... 13

4.2 Attributes and dimensions of incidents.
    process failure = pfail; outage = out; latency = lat; operation drop = drop;
    missing content = mis; perf = performance; func = functionality ........... 18
Chapter 1

Introduction

Companies and other organizations employ software systems necessary for their services and operations. Downtime for these systems can cause loss of reputation, customers, productivity and trust. To address this challenge, companies hire support engineers, going by various titles, to play an active role in supporting systems that must maintain high-availability. These engineers develop and maintain monitoring systems and tools to respond to system outages and service degradation by diagnosing and mitigating issues that arise, all with the goal of maintaining acceptable availability and performance.

Incident response is one among many responsibilities of a support engineer and it has many distinguishing characteristics. Barret et al. observed that the incident response is a peculiar task given it is ingrained with noticeable characteristics, including actions that may carry high risks, pressure, incomplete information in some cases and especially urgency for a solution [6]. They called for the HCI community to focus on system administrators as unique users, given their knowledge and tasks (including the incident response). The uniqueness of the incident response and its importance to organizations make it of interest to us.

Despite that importance, incident response is not well studied by the Software Engineering research community. However, it receives more attention in industry. For instance, there is an extensive list of commercial tools that can help support engineers while they monitor and respond to incidents.¹ The list of tools grows each year, with new emerging and existing tools receiving new features and functionalities, like PagerDuty, VictorOps and

¹The website alternativeto.net list 36 systems with the tag incident-management. Some other tags on the website that would apply are “monitoring”, “incident response” and “issue-tracking”.


BigPanda. Furthermore, industry reports that the management costs of a system (that includes incident response) far surpasses the cost of the software and hardware components of a system.\footnote{The report is available in “Autonomic Computing: IBM’s Perspective on the State of Information Technology”; http://www.ibm.com/industries/government/doc/content/resource/thought/278606109.html}

Few studies that are related to incident response contribute to the development of this subject. Listed among these studies and relevant to the topic of incident response are the description of the work of system administrators [2, 6, 9, 34], how they seek information during troubleshooting activities [17, 46], and how tools should be designed for them [23, 47]. Also worth noting is research on troubleshooting. Some of these studies focus on how to provide training that will help professionals obtain a better performance during troubleshooting [36, 38] and the implementation of tools that can give useful information to those who troubleshoot issues in systems [11, 15].

Missing from the literature is a theoretical foundation focused on incident response. In this study, we interviewed 12 support engineers, going by many titles. They shared 22 incidents in total, varying from a small latency to a complete outage. Using an approach based on grounded theory as explained by Corbin and Strauss [16], we analyzed each incident and created a model of the incident response for support engineers. We also identified the core dimensions of an incident and created an ontology for them. We validate the usefulness of our analytical results by proposing an improvement on monitoring tools. Lastly, we discuss important concepts that demand more attention from the software engineering community.

1.1 Definitions

- \textit{Incident}: is an unexpected problem that happens in a system. It can be described as an outage, service degradation, or anything that presents a risk and demands immediate correction.
• **Incident response**: is the process by which support engineers, along with other professionals, react to an incident. It begins when an incident is detected and ends when the incident is mitigated.

• **Support engineer**: is someone responsible for one or more systems that must take the needed measures to mitigate an issue. They can be called by other titles like software engineer, system administrator, site reliability engineer or technician, as long as they have the cited responsibility.

• **Mitigation of an incident**: is when a support engineer solves the problem or temporarily decreases or eliminates the negative effects of an incident to the point that there is no more urgency. Even once an incident is mitigated, there still may be additional work to address root causes, but that is not the focus of our research.
Chapter 2

Related work and background

In this section we will explore direct and indirect types of contributions towards the understanding of an incident response. Section 2.1 explores studies on heavy cognitive tasks related to programming and coding. Section 2.2 discusses exploratory studies related to troubleshooting and investigation of incidents. Section 2.3 examines studies that focus on tools used by support engineers. Section 2.4 shows a study related to training for incident response. Section 2.5 explains grounded theory as the basis of our methodology for this exploratory study.

2.1 Exploratory studies in heavy cognitive tasks

Although there are unique characteristics of incident response tasks, they resemble other heavy cognitive IT tasks given that the support engineer must deal with incidents with missing information and unknown causes in many cases. Heavy cognitive activities are common in many different tasks of software engineering. These activities demand a professional to understand something complex, like a source code, a system architecture or even a corpus of documentation, many times with some type of missing information. To help software engineers in these hard tasks, researchers have produced influential exploratory studies that delivered theoretical background to help future research.

For example, Ko et al. explored how programmers work during software maintenance tasks, exposed that developers spend 35 percent of their time in searches for their desired information and proposed a model to perform maintenance tasks [27]. Sillito et al., in an
observational study involving programmers during software evolution tasks, created the most comprehensive catalog of questions that programmers ask at that time [40]. Mayrhauser and Vans compared different models in the area of program comprehension and found limitations on the existing models, paving the way for the program comprehension community to build better models [48].

Although quite common in program comprehension tasks and other software engineering activities [39], we were only able to find 4 exploratory studies of support engineers responding to incidents [6, 9, 34, 47]. An exploratory study can help future research aimed at incident response. We talk about each of these studies next.

2.2 Studies on troubleshooting and information gathering and causes of problems

Bereiter and Miller [9] identified difficulties in diagnosing faults in automated manufacturing systems. They identified that errors could be present in the design of systems and structures, in software and communication dynamics. As a result, they provided different types of recommendations to help support engineers troubleshoot issues, as well as recommendations to prevent errors.

Another important task during incident response is the information gathering to solve issues. Past studies explored this process in a generic way. In one such study, Barret et al. observed working patterns from system administrators [6]. Their work explained that, while responding to incidents, support engineers work vigorously to understand causes of an incident. It also described that they must resort to many different channels to obtain information. In their reported cases, sometimes a support engineer can obtain the necessary information using their own tools, while other times it needs the expertise from other professionals, including results from tests that only these professionals can perform. They learned that system administrators use 11% of their time troubleshooting. They described communication dynamics. Their data suggested that their work is as much social as technical (90% of
troubleshooting work was a human-to-human interaction), and that they lose on average 25% of their time on diversions, like searching for someone that can grant access to a system in order to make a test.

On another study, Bystrom and Jarvelin observed the correlations between task complexity and information gathering [12]. Their findings suggest that as complexity increases, the complexity of the information increases, more domain and problem-solving information is needed, and the success rate of seeking information decreases. A more recent study by Velasquez and Durcikova focused on the information gathering activities made by system administrators [46]. Their findings confirmed that system administrators, responsible for responding to incidents, will seek information to verify the results of the work they have done when task is considered complex. Adding to this are the findings of Patterson et al., showing that more hardware and software components make the troubleshooting a more complex and time consuming task [34].

In another study, Souza et al. discovered that tools used during troubleshooting have the required information in just 33% of cases, and that support engineers must consult to other professionals including customers, software administrators and people responsible for monitoring and handling of critical situations [17]. They described how long it takes to get the right information, as well as how many sources a support engineer must consult to troubleshoot something.

### 2.3 Studies on tools to help support engineers

Support engineers rely greatly on information that comes from their tools and from other professionals. As Velasquez and Weisband observed, given the technical nature of the tasks performed by system administrators, it is impractical to talk about their work without talking about their tools [47]. Given this, we believe that tools play an important role during incident responses, be it by automating responses, monitoring systems or providing information.
Bhatia et al. implemented a tool that collects information directly from the operating system to provide useful information to be used by support engineers during incident responses [11]. They formulated, through the combination of the information provided by their tool, a set of rules to help support engineers during the diagnosis process. In another study, Chiarini argues that a support engineer must understand the environment of systems based on a precise mental model that represents the system [15]. To help support engineers, he demonstrated that it is possible to understand the dependencies among different systems using arguments from the kernel data structure, thus giving a real representation of the system dependencies to help support engineers.

Haber and Bailey explored the ethnographic aspects of a system administrator to design general guidelines to create tools that support system administrators [23]. Their field studies corroborated other studies that observed that tools are not well aligned with work practices. They observed that poorly designed tools could also cause problems or lengthen a resolution. Another design study by Velasquez and Weisband considered system administrators, also responsible for troubleshooting, as a distinct group of users with considerable differences in background and tasks [47]. They contributed to other studies by identifying attributes from systems and from information that are desirable in a tool designed for a system administrator.

One reason we believe that a theoretical foundation may contribute to research in incident response is that, as Suchman argues, designers must consider the context to create an effective system [44]. Thinking this way, tools designed to help support engineers and system administrators to respond to incidents must consider the context to be effective.

2.4 Studies of training as an approach to help support engineers

A different approach to help support engineers is to focus on their workflow process rather than their tools. For instance, Schaaifestal et al. showed the importance of training in troubleshooting, where technicians trained in structured troubleshooting solved incidents from naval ship systems using 50% less time than technicians without training [38]. They observed
that there is “a gap between theoretical knowledge and the application of this knowledge in real-life situations”. They also noted that a highly structured system representation is important to help technicians troubleshoot.

2.5 Grounded Theory

As stated before, the methodology in this proposal is based upon the grounded theory methodology. Created by Glaser and Strauss, grounded theory is a systematic method designed to generate theory from data [20]. It is clearly different from the standard scientific method where a hypothesis is first formed to then be tested. Adolph et al. explain that Glaser and Strauss use the word “grounded” because the theory is obtained by following a systematic and rigorous process of constant comparison of data (normally indications of or incidents related to a concept), thus resulting in a theory “grounded” on the data [1]. Adolph et al. also explained that the grounded theory method is useful for research in areas that were not studied or when a new perspective might be beneficial. Grounded theory is usually inappropriate to test hypotheses.

Adolph et al., among many authors, provides an overview of grounded theory [1]. A researcher interested in studying a phenomenon collects data, that could be interviews, observations, videos, diaries, newspapers and other sources according to Corbin and Strauss [16]. The researcher will analyze the data, searching for concepts in the incidents that will be assembled and turned into categories. To develop the theoretical properties of these categories, incidents from arriving data are compared to incidents from previous data regarding a category. This process continues until Theoretical Saturation is achieved, meaning that the analysis of incoming data will not result in new concepts, and existing categories are well developed. When available, the resulting theory may be compared to theories from the existing literature. During all phases of this process, the research team writes memos containing the tactics, judgment and logic behind the emergence of concepts, categories and their relationships. For
Corbin and Strauss, the resulting set of well-developed categories systematically interrelated constitutes the theory generated by this methodology [16].

It is useful to note that there are different versions of grounded theory. Ralph et al. list four of them: the first version by Glaser and Strauss (1967), a later version by Strauss and Corbin (1990-1998), a constructivist version by Charmaz (2000-2006), and a more recent version by Birks and Mills (2011) [35]. Our methodology in this study is based upon the methodology explained by Corbin and Strauss, that, according to Adolph et al., is more structured and often used by software engineering researchers [1].
Chapter 3

Thesis Statement

The goal of this research is to create (1) an ontology of incidents and (2) a model of the incident response. We claim that our analytic results will help with the design of (1) better monitoring tools, (2) incident response plans and training, and (3) robust systems better prepared for incidents.

To validate the quality and validity of our work, we could use our developed ontology and model to identify and propose opportunities to improve all these 3 proposed design improvements. However, given the scope and time restraints, we limited our validation of the ontology and model by proposing a design of an improved monitoring tool. Improvements will be designed to solve deficiencies present in our data. In section 5.1 we discuss the ontology of incidents. In section 5.2 we discuss the model of the incident response. In chapter 6 we validate our work by proposing a monitoring tool improvement based on some key observations as stated on section 5.3.1, 5.3.2 and 5.3.3.
Chapter 4

Methodology

To achieve our goal in this qualitative study, we used an approach based on Grounded Theory as explained by Corbin and Strauss [16]. In our methodology, we performed two main activities: the collection of data and its analysis. Different from other methodologies, data collection and data analysis are interleaved activities, being two faces of the same process. This implies that we did not collect all data before beginning our analysis.

We interviewed support engineers, asking about their experience dealing with incidents. We transcribed the interviews and analyzed our data. We identified relevant attributes of and their dimensions from our data, using them to build a model of incident response and an ontology of incidents. In what follows, we will describe how we collected data for our model and ontology. Afterwards we will describe the data analysis process.

4.1 Data Collection

We collected our data by interviewing support engineers using a semi-structured style. (See appendix A for the interview guide we used.) We interviewed 12 support engineers from 7 companies, obtaining in total 22 incidents. The reported roles of the participants in this study are software engineers, site reliability engineers, managers, and system administrators. The list of participants and incidents can be found in table 1. (See appendix B for a more detailed description of each incident.) All interviews were recorded and later transcribed. This preserved everything that was said for future analysis. Interviews were done in person or through an online tool (Skype or Google Hangouts). In total, 5 interviews were conducted.
in person and 7 were assisted by an online streaming tool. All transcripts were anonymized, meaning we removed information that would identify a participant or a company.

In Grounded Theory, a concept is the unit of analysis [20]. The units of analysis in this study are the incident and the actions taken in response. We asked support engineers to describe their experiences dealing with these incidents, from the discovery of a problem to its mitigation.

We adapted and changed our interview focus as we identified concepts, attributes and dimensions that needed to be explored. We also searched for engineers working at different companies, with different backgrounds and experience levels. By changing our interviews to better understand such missing concepts and by searching for engineers with the desired experience, we applied what is called theoretical sampling [22]. This technique was applied to this case, resulting in variety in our data set, which implies a richer source to build the proposed model. One example is the concept of verification of a mitigation, something that was identified later during analysis after a few interviews and was incorporated later to future interviews.

Another technique that we applied during the data collection process is called member checking [28]. After we identified important concepts missing from previous interviews, we asked our participants to provide further data on some subjects. This allowed us to compare incidents and build a better model of the incident response. One such example of a concept we followed up on was about how they verified that a mitigation was successful, something that was missing from our initial interviews.
<table>
<thead>
<tr>
<th>Incident</th>
<th>Name</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.I1</td>
<td>Unassisted website upgrade</td>
<td>Upgrade of website to a new design failed during the night, resulting in a defective website</td>
</tr>
<tr>
<td>P1.I2</td>
<td>Docker no stress relief</td>
<td>Docker saving old image files without eviction caused disk space to fill and websites stopped working</td>
</tr>
<tr>
<td>P1.I3</td>
<td>Denial service attack</td>
<td>Denial service attack overwhelmed server and resulted in high latency on websites</td>
</tr>
<tr>
<td>P2.I1</td>
<td>Blocked DB</td>
<td>Blocked DB delete operation blocked processes of a system</td>
</tr>
<tr>
<td>P2.I2</td>
<td>Synchronous and under scaled</td>
<td>System shows a timeout error because server is under scaled and communication is synchronous</td>
</tr>
<tr>
<td>P3.I1</td>
<td>Random error on external API</td>
<td>Users were unable to connect to service because external API Rare and random API</td>
</tr>
<tr>
<td>P4.I1</td>
<td>Changing one output</td>
<td>Code change resulted in more and smaller output files, breaking receiving machine learning</td>
</tr>
<tr>
<td>P4.I2</td>
<td>No smoking gun</td>
<td>Undetected error on load balancer resulted in lost revenue, just discovered in final metrics</td>
</tr>
<tr>
<td>P5.I1</td>
<td>Changing DB usage</td>
<td>Recent change began to use solely DynamoDB without upgrading maximum usage, resulting in under scaling error</td>
</tr>
<tr>
<td>P5.I2</td>
<td>Poisoned cache</td>
<td>Change in code resulted in poisoned cache unreadable by external services</td>
</tr>
<tr>
<td>P6.I1</td>
<td>Time bomb</td>
<td>DB reached maximum key size, resulting in errors</td>
</tr>
<tr>
<td>P6.I2</td>
<td>Welcoming present</td>
<td>Data center dropped during holiday, and on-call experienced engineer was new to company</td>
</tr>
<tr>
<td>P6.I3</td>
<td>Broken by tests</td>
<td>New Kubernetes cluster began to fail to create new pods because IP addresses were burned during test</td>
</tr>
<tr>
<td>P7.I1</td>
<td>Misleading incident</td>
<td>System ongoing errors were interpreted as symptoms of an incident, misleading engineers</td>
</tr>
<tr>
<td>P8.I1</td>
<td>Log growth</td>
<td>Code changed to log ceaseless action, resulting in an increased pace on disk space usage</td>
</tr>
<tr>
<td>P8.I2</td>
<td>Old configuration file</td>
<td>Deployment of a very old configuration file resulted in outage and abnormal function of some systems</td>
</tr>
<tr>
<td>P9.I1</td>
<td>NTP error</td>
<td>Cluster lost quorum because removal of one machine changed NTP configuration from its dependents</td>
</tr>
<tr>
<td>P10.I1</td>
<td>Obsolete module</td>
<td>App deployed was using outdated authentication module, resulting in random error</td>
</tr>
<tr>
<td>P11.I1</td>
<td>Cache eviction</td>
<td>User unable to find product because cache eviction and system was unable to read from backend server</td>
</tr>
<tr>
<td>P11.I2</td>
<td>Waiting for a solution</td>
<td>Services stopped working because physical event dropped data center for some hours, and engineer had to await</td>
</tr>
<tr>
<td>P12.I1</td>
<td>Cascade of events</td>
<td>Sudden increasingly broadcast of network configuration culminated in latency, partial outage and complete outage</td>
</tr>
<tr>
<td>P12.I2</td>
<td>Failover failure</td>
<td>Firewall failed, and failover for firewall also failed</td>
</tr>
</tbody>
</table>

Table 4.1: Attributes of incidents
In a perfect scenario, we would continue to interview and analyze the data until we obtained theoretical saturation. Morse defines Theoretical Saturation as the phase where the researcher, after continuous sampling and analysis of the data, stops finding new concepts and all concepts in the theory are well developed [32]. However, we recognize that the creation of an exhaustive ontology and model representing all possible existing incidents may be a formidable and long endeavor. Because of practical considerations and limitations on time and resources, we did not achieve this goal. This means directly that our ontology and model may present incompleteness errors [21]. During the ontology and model creation, we will avoid the creation of any inconsistency and redundancy errors, since these errors do not depend on completeness.

We excluded from this study 8 incidents that lacked enough details about the incident and the actions taken. Participants were never forced to reveal data that they weren’t comfortable sharing. We also excluded 3 security incidents. These incidents are about a possible invasion on their systems. After some analysis, we identified that the nature of those incident responses is very different from the incidents in this study. In total, 3 participants and 11 incidents were excluded. The remaining 12 participants and 22 incident response cases were used in our analysis.

### 4.2 Data Analysis

Our first analysis began during the transcription process. Merriam et al. say that transcription helps researchers become intimate with the data and is an opportunity to write analytical memos, capturing some possible important concepts [31]. Another author, Wengraf, also encourages this idea by noting that memos are a product to be expected during the transcription process [49].

Memos, representing the tactics, judgement and logic behind emerging concepts were written at all stages of this study. Memos, as an echo of the analytical thought, represent ideas, questions, strategies, judgement, and logic behind the emergence of concepts used in
this study. The excerpt below illustrates a memo that was used to define the concept of symptom and root cause:

What someone may call a symptom, another may call a cause. For instance, a team from P4 discovered the problem was a larger than expected number of files being generated. For another team (the one responsible for the system) the number of files generated was a symptom of another problem.

For this study, three types of coding were used: open, axial and selective. We will explain each one of them and how they were used in this study.

*Open coding* is the phase where we identified, labeled, categorized and described aspects of the unities of our analysis (incidents and the incident response). Some examples of codes found in our analysis were ‘information gathering’, ‘learning’, ‘detection’, ‘handing off’ and ‘notification blast’. We iterated through all the codes of this study multiple times to ensure that coding was consistent, which means that all concepts that meant the same thing were using the same label. After this unification of codes, we re-coded each incident. By comparing codes in different incidents, we identified dimensions and attributes necessary for our analysis. For instance, after a brief comparison of the code ‘notification’ in all incidents, we identified that a notification could be manual or automatic, thus creating a dimension for this code. The same analysis was done for all identified codes.

*Axial coding* is the phase where we linked different codes and found patterns among them. We focused on causal relations rather than comparing all existing codes. One example is comparing ‘detection’ with ‘time for mitigation’, with their multiple dimensions. In this example, we were able to identify that there was always some type of gap between initial impact of an incident and manual detection, thus resulting in a longer mitigation time. As a result of axial coding, we understood not only each individual concept, but the relation among concepts.
Selective coding is the phase where we identified the core concept of our data. In this study, it means the most important attributes of incidents and the most important actions of the response.

Another technique used as part of our analysis is diagramming. Drawing diagrams help conceptualize data in a clear and related way. Corbin and Strauss argue that diagram drawing compel researchers to understand the data in a manner that reduces the data to their essence [16]. We drew multiple diagrams representing our understanding of the incident response based on our data alone. The diagram improved as more incidents were added to our data. One final diagram we created represents the model of the incident response and is included in this thesis in section 5.2.

Ontology of incidents

As described above we identified and explored attributes and dimensions related to each incident. For instance, one identified attribute of an incident was how long it took for its mitigation. Following this example, the length of the response is the dimension of such an attribute. In our study, attributes were not predefined, since our goal was to build a theory, not test one. The initial objective of our analysis was to identify as many attributes as possible among our incidents. After some interviews, we identified what attributes were important to our objective. We were able to compare incidents and discern the most relevant attributes based on their dimensions. Dimensions with a single attribute among different incidents contributed little to our analysis and were dropped. We have interest in their differences. The defining attributes will be used to classify, explain and understand incidents. This explain why we searched for variety in our sampling.

Model of incident response

We also analyzed how support engineers respond to incidents. We identified every action that a support engineer took in response to an incident along with its attributes and dimensions. Just as an example, one type of action that a support engineer performed was 'learning'. We identified attributes of this action, like what sources an engineer used. In this
example, exploring the attribute source, the dimensions of this attribute were 'expert', 'search engine' and 'documentation'. The attributes and their dimensions were derived from the data rather than a predefined list. As we identified attributes and dimensions that required more exploration, we gathered more specific data if possible.

Each incident response process is comprised of several actions. They were studied in isolation and in groups. For each incident response, these actions were structured chronologically, from detection of an incident to the verification of a mitigation. Logical connections between different actions were analyzed. The incident response processes were organized and described at a high level. After a few interviews, we began to create our model that represents the incident response processes. The initial model was created by comparing the incident response processes in our data set. The model was able to represent each different incident response process in our data set at the time. As we added incidents to our data set through subsequent interviews and noted that the model was not able to fit the new data. Whenever it was needed, we modified the model until it represented well all 22 incident responses in our data set.
<table>
<thead>
<tr>
<th>Incident</th>
<th>Name</th>
<th>Symptom</th>
<th>Impact</th>
<th>Detection</th>
<th>Hand off</th>
<th>Mitigation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.I1</td>
<td>Unassisted website upgrade</td>
<td>Specific: pfail</td>
<td>Functionality</td>
<td>Manual: late</td>
<td>1</td>
<td>Roll back</td>
<td>4h</td>
</tr>
<tr>
<td>P1.I2</td>
<td>Docker no stress relief</td>
<td>Generic: out</td>
<td>Functionality</td>
<td>Manual: late</td>
<td>1</td>
<td>Free up resources</td>
<td>+2h</td>
</tr>
<tr>
<td>P1.I3</td>
<td>Denial service attack</td>
<td>Generic: out</td>
<td>Performance</td>
<td>Automatic: on-time</td>
<td>0</td>
<td>Change system</td>
<td>1.5h</td>
</tr>
<tr>
<td>P2.I1</td>
<td>Blocked DB</td>
<td>Generic: lat</td>
<td>Functionality</td>
<td>Automatic: on-time</td>
<td>0</td>
<td>Change system</td>
<td>+1h</td>
</tr>
<tr>
<td>P2.I2</td>
<td>Synchronous and under scaled</td>
<td>Specific: pfail</td>
<td>Perf + Func</td>
<td>Manual: late</td>
<td>0</td>
<td>Change system</td>
<td>+8h</td>
</tr>
<tr>
<td>P3.I1</td>
<td>Random error on external API</td>
<td>Specific: pfail</td>
<td>Functionality</td>
<td>Manual: late</td>
<td>0</td>
<td>Change system</td>
<td></td>
</tr>
<tr>
<td>P4.I1</td>
<td>Changing one output</td>
<td>Specific: pfail</td>
<td>Functionality</td>
<td>Manual: late</td>
<td>1</td>
<td>Fix code</td>
<td>3d</td>
</tr>
<tr>
<td>P4.I2</td>
<td>No smoking gun</td>
<td>Generic: drop</td>
<td>Performance</td>
<td>Manual: late</td>
<td>5+</td>
<td>Scale up</td>
<td>+1w</td>
</tr>
<tr>
<td>P5.I1</td>
<td>Changing DB usage</td>
<td>Generic: lat</td>
<td>Performance</td>
<td>Automatic: on-time</td>
<td>0</td>
<td>Scale up</td>
<td>1h</td>
</tr>
<tr>
<td>P5.I2</td>
<td>Poissoned cache</td>
<td>Generic: lat</td>
<td>Performance</td>
<td>Automatic: on-time</td>
<td>1</td>
<td>Fix code</td>
<td>+12h</td>
</tr>
<tr>
<td>P6.I1</td>
<td>Time bomb</td>
<td>Specific: pfail</td>
<td>Functionality</td>
<td>Automatic: on-time</td>
<td>0</td>
<td>Scale up</td>
<td>20m</td>
</tr>
<tr>
<td>P6.I2</td>
<td>Welcoming present</td>
<td>Generic: out</td>
<td>Functionality</td>
<td>Automatic: on-time</td>
<td>0</td>
<td>Reinstall services</td>
<td>12h</td>
</tr>
<tr>
<td>P6.I3</td>
<td>Broken by tests</td>
<td>Specific: pfail</td>
<td>Functionality</td>
<td>Manual: on-time</td>
<td>0</td>
<td>Scale up</td>
<td>+4h</td>
</tr>
<tr>
<td>P7.I1</td>
<td>Misleading incident</td>
<td>Generic: drop</td>
<td>Functionality</td>
<td>Automatic: on-time</td>
<td>1</td>
<td>Fix configuration</td>
<td>7h</td>
</tr>
<tr>
<td>P8.I1</td>
<td>Log growth</td>
<td>Generic: lat</td>
<td>Performance</td>
<td>Automatic: on-time</td>
<td>0</td>
<td>Free up resources</td>
<td>+1.5h</td>
</tr>
<tr>
<td>P8.I2</td>
<td>Old configuration file</td>
<td>Generic: out</td>
<td>Functionality</td>
<td>Automatic: on-time</td>
<td>1</td>
<td>Fix configuration</td>
<td>3h</td>
</tr>
<tr>
<td>P9.I1</td>
<td>NTP error</td>
<td>Specific: pfail</td>
<td>Functionality</td>
<td>Automatic: on-time</td>
<td>1</td>
<td>Fix configuration</td>
<td>4h</td>
</tr>
<tr>
<td>P10.I1</td>
<td>Obsolete module</td>
<td>Specific: pfail</td>
<td>Functionality</td>
<td>Manual: late</td>
<td>1</td>
<td>Change system</td>
<td>2w</td>
</tr>
<tr>
<td>P11.I1</td>
<td>Cache eviction</td>
<td>Specific: mis</td>
<td>Functionality</td>
<td>Manual: late</td>
<td>0</td>
<td>Scale up</td>
<td>+12h</td>
</tr>
<tr>
<td>P11.I2</td>
<td>Waiting for a solution</td>
<td>Generic: out</td>
<td>Functionality</td>
<td>Automatic: on-time</td>
<td>2</td>
<td>Restart services</td>
<td>1h</td>
</tr>
<tr>
<td>P12.I1</td>
<td>Cascade of events</td>
<td>Generic: lat</td>
<td>Perf + Func</td>
<td>Manual: on-time</td>
<td>0</td>
<td>Change system</td>
<td>6h</td>
</tr>
<tr>
<td>P12.I2</td>
<td>Failover failure</td>
<td>Generic: out</td>
<td>Functionality</td>
<td>Manual: on-time</td>
<td>0</td>
<td>Restart services</td>
<td>15m</td>
</tr>
</tbody>
</table>

Table 4.2: Attributes and dimensions of incidents.
process failure = pfail; outage = out; latency = lat; operation drop = drop; missing content = mis; perf = performance; func = functionality
Chapter 5

Study Findings

This chapter will present our findings divided in three sections. In section 5.1, we discuss meaningful attributes of incidents. In section 5.2, we discuss the resulting model of incident response representing our data. In section 5.3, we discuss the key observations of this study, namely important concepts to improve the incident response process.

5.1 Incidents

After initial analysis, we identified 6 attributes of incidents that are relevant for the goals of this study. For each incident, we identified their (1) symptoms and starting point, (2) root cause, (3) impact, and (4) time to mitigation. The following sections describe these attributes and their dimensions.

5.1.1 Symptoms and starting point

Starting symptoms are the first signals disclosed to the support engineer. Different symptoms may be identified as the investigation progresses. One important dimension is symptom specificity. On one side we have generic symptoms, while on the other side we have specific symptoms. We identified 9 incidents with a specific starting symptom and 13 incidents with a generic starting symptom.

1. Generic symptom: a symptom is defined as generic when it can be the result of many different root causes. For instance, latency can be a result of different issues, like lack of disk space (e.g. P8.I1), limited processing power (e.g. P1.I3), or gradual failure of a
component (e.g., P5.I2). We consider these symptoms as 1-N, meaning one symptom that can be caused by many different root causes. The following are examples of generic symptoms:

- Outage: complete unavailability of a system or a functionality. For instance, in P1.I2, a group of websites became unavailable for their users.
- Latency: unacceptable slowness and response time. One example is P1.I3, where a group of websites were taking more time than desired to download and show their content.
- Drop in operations: degraded performance. In P4.I2, a higher-up noted a slight drop in operations, thus starting an investigation.

2. Specific symptom: a symptom is defined as specific when there are only a few possible root causes. One example is P1.I1, where a specific website expected to update their design failed, showing an incomplete and strange design and behavior. Another example is P2.I2, where a user reported a time-out event while trying to create a streaming. These symptoms narrow down the scope of the investigation, providing support engineers a hint of where to start.

Symptoms are an important part of the troubleshooting process and the starting point. An engineer may identify different symptoms as the investigation progresses. We will discuss some relevant dimensions for symptoms.

One important dimension of symptoms observed in our data is how close a symptom is to a root cause. A symptom can be close to a root cause or be a distant side effect. One example of a symptom distant to a root cause is P4.I2, where a metric generated by the data team was showing a degraded performance in one operation. The metric was the final output of a multi-stage data processing pipeline, each stage owned by a different team. The data team began their investigation, only to discover that their data processing was correct. They handed off the incident to the team that owned the system that was providing the input data. This pattern repeated five times until a team discovered a problem in the load
balancer. Another example of a root cause closely related to the symptom is P6.I1, where the support engineer received a specific type of database error multiple times. Although one metric showed the database to be fully functional and working, the engineer was able to quickly check the code trying to talk to the DB. The engineer was also able to check the DB. By replicating the error, the engineer was able to identify a root cause and mitigate the issue. Detection is discussed in section 5.2.1. The distance of a symptom to its root cause is explored in section 5.3.3.

5.1.2 Root cause and mitigable point

Our initial concept of root cause was an event that triggered the whole chain of events culminating in the incident. We changed our definition after careful analysis of our data. While an incident can be explained by a single event, our data shows that an incident is the culmination of many different events and behaviors.

To explain these concepts, we will create a chain of events that links a symptom of an incident to events and behaviors related to it. We will demonstrate this by taking P9.I1 as an example. Starting with the symptom, we ask ‘why’ it happened. We will continue until no more answers are provided. In P9.I1, the starting point was quorum loss, result of machines being unable to find consensus through vote, result of different time difference between machines, result of NTP configuration change, result of removal of machine from cluster that other machines were reading to get NTP. We could continue this chain with a hypothesis like ‘machine was removed because circuit fried’, ‘because circuit breaker was not working as expected’, ‘because support forgot to replace it’ and ‘because it was old’, ‘because...’ and so on. This is similar to what Lions did, during the report of Ariane 5 rocket launch failure, describing the error by explaining a chain of technical events that resulted in the failure rather than pointing to a single event [29].

To find the complete chain of events and behavior that lead to an incident is not useful when an engineer needs to mitigate the incident as fast as possible. Rather than digging down
to find all existing causes in the chain of events that resulted in the incident, the support engineer needs to find one fixable point that will solve the issue at hand and, when possible, prevent the recurrence of the issue permanently. Further investigation of this chain of events in the form of a post-mortem could explain in a more detailed way how an incident happened. In our data, however, investigation during an incident stopped when one or more mitigable causes were identified. (A chain of events for each incident is available at Appendix B).

Figure 5.1: How P9.I1 happened by linking events and behaviors.

This mitigable point could be called a root cause. Note that there may be more than one root cause, meaning more than one mitigable point. For instance, TapRoot\(^1\) defines root cause as one or more than one basic causes that can be fixed and will prevent or reduce the

\(^{1}\)The website https://www.taproot.com/definition-of-a-root-cause/ discusses about different meanings of root cause and their implications. The definition that most resembles our observations is the second definition.
likelihood of the problem’s recurrence. In practical terms, it is very similar to what a support engineers tries to find during an incident response.

For this study, we call root cause a mitigable point rather than a single event that triggered the whole incident. We recognize that it is possible for an incident to have more than one root cause, or mitigable points. One example is P2.I2, where users began to receive timeout errors. The engineers looked at logs and discovered a slow response time that was triggering the error. The error was certainly the result of a combination of factors, like a higher number of users, synchronous communication and a server with scaling problems. Recognizing that “the interface between [the system] and the cluster was faulty” and knowing that fixing the open source software with scaling issues would be costly, his team mitigated with a workaround, changing the communication with the cluster from a synchronous to an asynchronous protocol, removing the possibility of the timeout error. This specific error would not repeat anymore, although latency would continue. The engineer could try to add another server or fix the open-source scaling issue. The engineer, nonetheless, had to work with his available resources to solve the issue at hand.

Although some human errors or bad practices can be considered one cause of an incident, they are not root causes, since rectifying a behavior or practices will not mitigate the problem at hand. It can help prevent an error in the future, but not solve it immediately.

How it happened

Chains of events, as explained, recount how one incident happened. One common element of these chain of events is that they show at least one change in behavior or in the environment.

Although the result of a change in behavior and environment, an incident may not be the result of a recent change in the system, like a deployment. An incident can be triggered by a defect or limitation on the system that, without the proper treatment, will one day result in one incident, like a time bomb. For instance, in P6.I1, a system was functional and healthy until a table reached the maximum key size, thus resulting in errors. Although healthy for
many years, the system was prepared to work for a specific amount of time. One could look at these forgotten details as time bombs since the system would hit the maximum key size at one point. Another example is P11.I1, where the provisioned cache had enough space for its operation given a particular number of content. Natural growth over time resulted in more content, unable to fit in cache, thus triggering the incident. Since the system was not designed to read from the back-end server and cache size was static, we can assume that this issue is also a time bomb. The system was fully functional during tests and production for some time. But a system working in tests in one environment has no warranty that it may be functional for future cases and in a different environment. As one engineer said, their problem was “something we hadn’t thought off”, and “it was just a matter of time, I guess.” (P11) An engineer must be aware of behavioral changes as well as environmental changes, like natural growth of number of users. Examples of incidents in our data that represent a type of time bomb, without any recent change related to the incident, are P1.I2, P2.I2, P4.I2, P6.I1, P6.I3 and P11.I1. We discuss more these types of incidents in section 5.3.4.

During an investigation, one engineer will try to find a mitigable cause to solve the issue. The complete chain of events, on the other hand, can explain how an incident happened and can help engineers prevent repetitions of the same incident in the future. Such process is recorded in documents called post-mortems. Reel says that engineers are doomed to repeat their errors if they do not perform a post-mortem analysis, including the analysis of how one incident happened [37].

5.1.3 Impact

Every incident in our data shows some sort of impact. This impact was always a threat to the mission of the system or mission of a company. During an incident, the support engineer tries to decrease or eliminate the impact, as one engineer well explained that his objective was to “[not] let the impact to customers [keep] existing.” (P5) Outage, latency, missing content or degraded functionality are examples of symptoms that wield some impact, meaning that
they are a threat to the mission of a system. We identified the following two dimensions of
an impact:

- Degraded performance: when a system is functional, but performance is limited. One
  example is P5.I2, where a poisoned cache resulted in more calls to the back-end, thus
  resulting in increased latency. The system was still functional.
- Degraded functionality: when a system or part of a system does not work as expected.
  For instance, in P6.I3, a cluster was operable, but the creation of new pods was failing
  given that all allocated IP addresses were depleted.

All incidents in our data are characterized by at least one major impact. Also, all
initial symptoms in our data represent some sort of impact. There is no incident without at
least some sort of threat to the mission of a system.

An incident can have both degraded performance and degraded functionality. One
example is P2.I2, where users started to receive timeout error messages while trying to access
a streaming service. This timeout happened because the server was under scaled, slowing
response time. This resulted in timeout errors. In this example we observe both degraded
performance (under scaled server) and degraded functionality (unable to stream one service).

A support engineer should also be aware that it is possible that a system can deteriorate
or change as time flows given the nature of the incident. Some incidents can stay in a static
state, some incidents can deteriorate with time, and some incidents can vanish with time. One
example of an incident that can degrade is P12.I1. In this incident, degraded performance
in the form of latency was the first identified symptom. After some time, one server went
down. It was not long enough that the remaining servers also dropped, thus resulting in a
complete outage. The incident began with degraded performance, ending with an outage,
considered a degraded functionality. Another case of an incident that can degrade is P5.I2,
where a poisoned cache resulted in latency. Had not the engineer mitigated the problem fast,
the load on the back-end server would become too much to be handled, thus resulting in a
worse impact.

We understand that not every impact results in an incident. A defect, a leak or any other system failure will only be considered an incident when there is a substantial impact on the system. For instance, a system may have multiple resource leaks that degrade the performance without such being considered an incident. Taking P7.I1 for an example, there were constant communication errors from system A to B. System B had many ongoing defects and failures. Nonetheless, the alarm only soared when a certain threshold of communication failures was exceeded. It does not mean that these ongoing defects, leaks and accepted failures will not influence the investigation. Continuing with P7.I1, the ongoing defects and errors on system B were interpreted as misleading signals by the support engineer team, prompting them to spend more time on system B, while the root cause was at system A.

5.1.4 Time to mitigation

The total time for mitigation can be calculated as time to detect an incident plus time to respond to it. Table 4.2 shows the total time to mitigation for each individual incident in our data. In our data, incident response time ranges from 15 minutes to 2 weeks with a median of 3.5 hours. Time to mitigation is important in our analysis given our main objective is to decrease time to mitigation.

Time to detection can play an important role in total time to mitigation. For instance, in P4.I2, a higher up noted a decrease in sales, rather than following a trend. After investigation, support engineers were able to identify a under scald load balancer, resulting in sale losses. As the engineer said, they made a query and discovered that the “count suddenly drops 4 or 5 days ago.” This late detection increases time to mitigation by 4 or 5 days. A second example is P10.I1, where access to a system was randomly failing 5% of times. The
engineer was notified by a core user that was notified by another user. This communication itself represents added time to mitigation. A third example is P4.I1, where the machine learning team discovered that their machine learning model had not updated in 24 hours. If the error was automatically detected, the mitigation time would decrease by 24 hours. We noted that time to mitigation is decreased when detection is fast, and will explore this in section 5.2.1.

5.2 Incident Response Model

Figure 5.2 illustrates the resulting incident response model of this study. It was built to reflect our existing data. Appendix B shows the response steps for each incident in our data. The main actions in this model are (1) detection, (2) notification, (3) investigation, (4) mitigation, (5) verification, and (6) handing off.

![Incident response model](image)

Figure 5.2: Incident response model.

5.2.1 Detection

Detection is the first acknowledgment that there is a possible impact present in the system. It is followed by a notification to an engineer. They are divided into two main groups: automatic detection and manual detection.
1. Automatic detection: monitoring tools that actively notify engineers when an issue is present.

   (a) By threshold: when the incident is detected by a monitoring system that notifies an engineer given a breach to a threshold of a certain metric. One example is P5.I1, where a sudden latency pushed a notification to the engineer. Engineers need to decide whether a certain metric level is acceptable or not. Another example is P9.I1, where a monitoring system created an automatic ticket and notified a support engineer after a certain number of failures to reach consensus in a cluster.

   (b) By events: when the incident is detected by a monitoring system given a specific event. One example is P6.I1, where the support engineer received each database error individually. The difference between a threshold detection and an event detection is that, on the second one, a single event is necessary for the notification, while on the first one, a specific number of events is necessary to push a notification.

2. Manual detection: detection by humans. This detection can be assisted by monitoring tools. Nonetheless, it is a human who suspects something may be wrong, rather than a tool.

   (a) Internal detection: when the incident is detected by someone inside the company, like another team or even a higher up. One example is P1.I1, where a supervisor saw that one website failed to update to a new design. The supervisor called for the on-call engineer to fix the issue.

   (b) External detection: when the incident is detected by someone outside a company. One example is P3.I2. In this case, a random error was detected by a user trying to login to a system. The user opened a ticket for the IT. One characteristic of this type of detection is the oblivious nature of the user related to the systems and structure of the company.
As noted in section 5.1.4, detection time is important to decrease time to mitigation. We categorized the time to detection in the following way:

- Early: when an incident is detected before it has an impact on the system or user.
- On-time: when an incident is detected at the time of the first impact. For instance, in P6.I1, immediately after the very first error, the engineer was automatically notified by email.
- Late: when an incident is detected while impact is going on. For instance, in P3.I1, the engineer understood that the existing problem were affecting other users even before someone decided to report the error. In P11.I1, another engineer knew that their system was evicting content for some time before one user reported the error.

These attributes are important to our analysis, since one of our main goals is to decrease time for mitigation. Below we explain the subtle differences of early, on time, and late detection, and their importance for the incident response process.

![Latency over time](image)

**Figure 5.3:** Differences of early, on time and late detection. The red line is a threshold representing an impact.

The concept of early, on time, or late detection is based on impact. For instance, given latency and a service level agreement (SLA), a detection can be considered early if
it is detected before a system reaches a certain latency threshold. A detection would be considered on time when the latency threshold is achieved and an engineer is notified. And a detection would be considered late if no automatic monitoring tool is deployed and detection happens after the latency threshold is exceeded.

One example of late detection in our data is incident P4.I2. A company executive noted a decrease in sales in a general metric. After a long investigation, the engineers were able to detect that a failure in the load balancer was resulting in revenue loss for 4 or 5 days. If one detection tool grasped the first load balancer error 4 or 5 days ago it would be considered detection on time. If an engineer detected that the load balancer was close to reach its peak capacity and needed to be scaled up, it would be an early detection.

Detection on time does not mean that a system was healthy and free of defects before detection. It means that detection happened when an impact began. One example is P5.I2, where an engineer received an automatic notification related to high latency. A root cause for this problem was the deployment of a defect in the code that would turn future cache entries unreadable by external machines. The system was broken before detection, but impact was felt later because it would take some time for a considerable portion of the cache to become unreadable. As a larger portion of the cache became unreadable, more requests were sent to the server. Getting information from the back-end server is slower than retrieving information from cache, increasing latency for that system.

We do not have a case of an incident that was detected early. We believe that this happened because participants were asked to share incidents, and prevention may not be considered an incident.

5.2.2 Notification

A support engineer is engaged in an incident by a notification. This step happens when an incident is detected. It also happens as a result of a hand off. There are two main types of notifications: Automatic notifications and manual notifications. An automatic notification is
a notification by an automatic tool, while a manual notification is triggered by a human. For automatic notification, we identified the following tools in our data: pager, email and text message. For manual notification, we identified the following channels used: phone call, chat, ticket system, email, and in person.

5.2.3 Investigation

Investigation is the core part of the incident response model. Investigation was identified as a information gathering activity. We will discuss in this section the information being gathered and investigation strategies to find a root cause.

One dimension of the information being gathered is specificity. Specificity is about how unique or common is the information being gathered. We can look at two opposites in this spectrum: learning about the current behavior of a system vs learning about a technology in general. We call diagnosing the action of gathering information about the current state and behavior, and learning the action of gathering general or common information.

Examples of diagnosing and learning can be observed in our data. For instance, in P3.I1, a user reported being unable to access features from a web page. Unable to replicate the error, the engineer looked at Stack Overflow to see how PHP stores session data (general information). The engineer logged into the server and found the data file (specific information). He discovered that the data file was encrypted. This information led him to use a search engine and learn how to decode his data (general information). After that, the engineer decoded the data file and looked at it, finding it empty (specific information). In this example, information about how PHP stores session data and how to decode data are very common and generic, while information about what was inside the session data is very specific to this system and situation. Given the observed differences between searching for specific information and general information, we will discuss each part separately in this section.
Diagnosing

While troubleshooting, engineers in our data worked to find the needed information in order to mitigate the issue. Given a system and its many components, the engineer will pursue some strategies to find a root cause. We identified two different types of search strategies: systematic search and opportunistic search.

A systematic search strategy is an orderly and methodical search, where an engineer will begin the investigation with a starting point, or symptom, and continue investigating across the system until a root cause is found. This search can start with a component possibly related to the symptom. One example is P11.I1, where engineers began their investigation with the top of the runtime stack, until they reached the cache and found that it was evicting data. Another example is P4.I2, where a team of engineers responsible for the sales dashboards began to investigate a sudden drop on sales. They found out that their system was healthy. They handed off the incident to the team that owned the system that provided the input data used by their system. This pattern continued five times until a root cause was found at the load balancer.

An opportunistic search strategy, on the other hand, are attempts to identify symptoms or anomalies across the system related to an incident. This search can narrow down the investigation, potentially saving time. During a search, an engineer may be guided by an hypothesis or by a group of symptoms. Below are examples of each of these opportunistic strategies:

- **Guided by hypothesis:** in P11.I1, support engineer began investigating what was historically the probable cause given his experience and knowledge. After receiving a common type of ticket, the support engineer began investigating if the user used a system properly to input data. Improper input was a common cause of error at that time. He called those possible cases as “low hanging fruit”, explaining that “The low hanging fruits often can solve your problem.”
- **Guided by symptoms:** in P1.12, an engineer received a report that a number of websites were offline. The support engineer tried to ‘find the point of commonality’, looking at what the websites had in common. The engineer noted all websites were from a single web server. He then proceeded to investigate the web server. In P5.11, the support engineer received an alarm of high latency from one service. The support engineer looked at dashboards and found the approximate time that latency increased. He found a deployment temporally correlated to the latency increase. He then proceeded to investigate the code change.

Engineers in our data combined different strategies to find a root cause to mitigate. One example where all the mentioned strategies are present is P1.13. A monitoring system detected latency on a group of websites. Guided by symptoms and finding the point of commonality, the engineer discovered that all websites were hosted on a single server. Then, guided by experience and trying the low hanging fruit, the engineer looked at resource utilization to see if disk space was full. He discovered that disk space utilization was normal. Guided by symptoms, the engineer searched for a lead and found that CPU utilization was high. Continuing with a systematic search, the engineer looked at what was using the CPU, discovering that docker containers were using it the most. Continuing the systematic search, the engineer looked at the specific process, and found that it was requests from a single IP address. Guided by hypothesis that it was a denial service attack, the engineer found the IP origin, discovering that IP was from a compromised server. The engineer then blocked the IP address, solving the problem.

Every incident in our data that started with a generic symptom began with an opportunistic search. As explained in section 5.1.1, a generic symptom can be caused by many different causes (1-N). On the other hand, every specific symptom provided a clue to where to start investigation.
Learning

Learning is the active process of getting the needed knowledge for the investigation. In this model, learning differs from diagnosing given the objective of the action. While diagnosing focuses on the behavioral aspects of the system, learning focuses on information about relevant technologies. The following learning strategies were identified in our data:

1. Learning using web resources. In our cases, the engineer used search engines to get technical information. One example is P1.I3, where the engineer used a search engine to understand how to block requests from a specific IP. Another example is P9.I1, where an engineer used a search engine to get more information about NTP configuration.

2. Learning by using internal documentation. One example is P6.I2, where an engineer was new to a company. Given a data center outage, the engineer used the documentation to list the critical services that needed to be restored. He also used the documentation to know the architecture of each service. A difference between learning using web resources and internal documentation is that some information is not available on search engines, like the internal architecture of a system or the pool of services hosted in one data center.

3. Learning through an expert, like a senior engineer, owner of a specific system, or even external specialist. An example is P1.I2, where the engineer sent a text to the last responsible engineer of a specific system to get technical information about the technology being used and how it was internally implemented. According to our interviewees, when immediately available, it is the fastest way to get some information. As this same engineer noted, the domain expert “could have probably resolved the entire issue in about 15 to 20 minutes as a domain expert. But I had no domain knowledge.” Another example is P10.I1, where the engineer reached out to a senior engineer to help with investigation after being unable to find any lead about the issue.
The senior engineer began to participate in the investigation. They tried to reach out to an expert on one API, although the expert was unavailable at that moment.

Given the variety and complexity of systems and technologies, it may be unreasonable to expect an engineer to know everything, specially if the on-call engineer is responsible for a high number of systems. For instance, an engineer noted that he was “in charge of maybe 25 services across the organization.” He was able to interact with the services to the point that he learned the ecosystem and became “a subject matter expert on kind of that whole flow.” (P6) After changing companies and becoming responsible for more complex systems, he noted that this learning strategy was “not super scalable” because of the number and complexity of systems. In a complex environment of systems, learning may be essential to an incident response.

Although necessary, learning can take time. None of the 4 incidents with response time of 1 hour or less had a learning action. Engineers in our data made observations that the learning tasks were costly and took time. P1 noted he had “no doubt” that the most consuming task during P1.I2 was learning with the expert because the expert was not immediately available to answer questions and guide the engineer. P2 also noted during P1.I2 that the most consuming task was learning about an open source queuing system, understanding why it was slow.

5.2.4 Mitigation

We call mitigation the act of solving the problem or temporarily decreasing or eliminating the negative effects of an incident to the point that there is no more urgency. We identified the following mitigation strategies in our data:

1. Fix and roll forward: fix configuration, code or even functionality of a system.
   (a) Fix code: fix code to make system functional. One example is P5.I2, where a recent deployment poisoned cache. The senior engineer fixed the code and added new lines that would turn the poisoned cache readable.
(b) Fix configuration: fix configuration to make system functional. In P7.I1, system A began to fail to call system B. After some investigation, the second on-call team discovered a change on the configuration address. They fixed the specific address, solving the issue.

2. Change system: change how a system works to make it functional. One example is P12.I1, where the engineer discovered that a port was overusing the I/O input by broadcasting the network configuration. To mitigate the incident, the engineer disconnected the management port, mitigating the issue. Another example is P2.I2, where the engineers turned the communication with server from synchronous to asynchronous to avoid timeout communication errors. A third example is P3.I1, where the engineer discovered that external API was returning occasionally empty files from requests. The engineer changed the system to detect these errors and repeat requests until a healthy file was retrieved.

3. Roll back: to revert code or configuration to a previous state. In P1.I1, the engineer rolled back the website to an older but functional design.

4. Get resources: to get needed resources to ensure the system will be functional.
   (a) Scale up: add resources to ensure the system is functional. For instance, in P11.I1, the engineer discovered that cache size could not fit content. The engineer scaled up by adding cache space. Another example is P6.I1, where a DB reached its maximum key size, resulting in an error. The on-call engineer increased the key size, making the system functional once more.
   (b) Free up resources: free up resources to ensure the system is functional. One example is P1.I2, where the engineer removed old image files from Docker to free up space, mitigating the issue.

5. Restart services manually: restart a system to make it functional.
   (a) Re-run services: reboot system to make it functional. In P11.I2, a dropped data center resulted in some services not working properly. After the data center was
brought back, the on-call engineer verified that two services were unavailable. He rebooted the services, mitigating the issue.

(b) Re-install services: re-install service to make it functional. In P6.I2, the engineer and company was unable to bring a dropped data center back. The engineer transferred the critical services from the dropped data center to new data centers.

One dimension that can be explored is longevity of solution. While some mitigation measures can prevent similar incidents in the future, others are temporary repairs. For instance, in P3.I1, the engineer noted that the modification of the system resulted in no more incidents of the same type in the long term. Another example of a permanent fix is P4.I1, where a change in the code resulted in a more than desirable number of output files. The code fix was enough to ensure that the system would work without further incidents. In that case, the incident will not repeat unless some change occurs. On the other hand, the engineer in P1.I2 released disk space to make a server functional again. Unless a script or another future measure was deployed, the incident would repeat in the future. Another example is P6.I1, where a table reached its maximum key value. The engineer scaled up the maximum key size. This is an example of an incident that can repeat depending on the growth of the company, time and new maximum key size. It is important to remember that, while a permanent solution may be desirable, the main objective of an engineer during an incident response is to mitigate the issue fast. In this scenario, a temporary solution may be desirable.

An engineer may have more than one mitigation path to take. Faced with these options, the engineer must make a decision based on his judgment and the mission of the system. For instance, in P5.I1, the engineer discovered that they reached the provisioned capacity limit for DynamoDB. The engineer had two options: use another database or increase provisioned capacity for DynamoDB. Since there was a relevant experiment going on, P5 decided to increase capacity. He explained that “it’s just personal judgment at that point. [He] wanted to keep the experiment on because [he thought his] team wanted it”. In another incident, P2.I2,
a system began to show a timeout error because the server was taking more time to answer than expected. Faced with this issue, the engineer could improve the performance of the open-source queuing system, or implement an asynchronous communication pattern to that queuing system. The engineer decided to make the communication protocol asynchronous, a less time consuming solution compared to the other option.

We also noted that an engineer does not need to know a root cause to mitigate an issue. For instance, in P1.I1, the engineer was able to roll back the website to a previous working state without knowing why the update failed. As he explained, “in the moment it really didn’t matter what was causing [it], because there was a simple solution available that would get us through the weekend.” As Jones noted, the focus of an incident response is not to find a root cause [10]. The focus of an investigation should be to mitigate the incident to save users from the effects and prevent degradation of the situation. Nonetheless, we identified in our data that in some cases it is necessary to find a root cause to mitigate an issue. One example is P12.I1, where the management port of one server was broadcasting a huge file to other servers, using most of the internal network. This high I/O output resulted in high latency, resulting later on the outage of one server and the subsequent outage of all remaining servers. The engineer was unable to mitigate the issue without identifying a root cause. They “had to actually fix the I/O problem before [they] could bring anything else up”. After they found a root cause, they disconnected the management port which was broadcasting a high traffic of data, thus being able to mitigate the issue. On the other hand, an engineer can try a blind mitigation, meaning a mitigation attempt without knowledge of causes of an issue. For instance, P1.I1 is an example where the support engineer blind mitigated the issue, and P5.I2, P7.I1, P12.I1 and P12.I2 are incidents where the support engineer tried to blind mitigate the issue without apparent success.
5.2.5 Verification

After a mitigation task, the engineer will try to verify that the impact on the system is gone. We observed that engineers in our data will apply one or more of the verification strategies below:

- Asking users if the issue persists: one example is P3.I1. One user was unable to access the system even with proper credentials. The user reported the error. Unable to replicate the error and without access to the external API, the engineer asked the user to perform some actions, including logging out and logging in repeatedly to the system. Some instructions followed by the user resulted in the mitigation of the issue, and the engineer was able to verify that issue was mitigated by listening the feedback from that user.

- Replicating error: one example is P1.I3, where the engineer detected a denial service attack from a compromised server. After using IP tables to block the IP, the engineer accessed the server with the blocked IP and tried to send a request to the server. The request was blocked and so the engineer knew the incident was mitigated.

- Checking symptoms: checking dashboards, metrics, resources, logs, or the system itself to verify if initial symptom is present. One example is P8.I1, where a recent code change began to write an frequent event in the log files that resulted in an explosive log growth. After rolling back the code to previous state and deleting the resulting large log file, the engineer looked at logs to see what was being written, noting that the frequent event was being omitted from log. The engineer also looked at metrics frequently to see if disk space would continue to increase as before. Disk space was steady. Checking symptoms can also demand a waiting time, as in P10.I1, where the engineer “waited to see if there was any error”. He deployed a patch fix and monitored HTTP status for some time to ensure that issue was mitigated.
• Verifying system is working: one example is P12.I2, where the firewall failed and the fail over firewall also failed. After restarting the firewall, the engineer verified that their systems had access to the network.

As in mitigation and diagnosing, engineers can employ multiple verification strategies to their incidents. In P5.I1, the engineer verified the mitigation by (1) looking at a decrease on time-to-call metric, (2) checking that CPU metrics returned to normal and (3) sending manual requests to see if an error would return.

According to our data, an engineer attempting to mitigate without a clear knowledge of the problem and root causes, may mistakenly assume that an issue is solved or persists by mistakenly verifying the symptoms. One good example is P5.I2, where the first on-call engineer observed that a recent deployment was temporally correlated to an incident. After rolling back that deployment, the engineer looked at metrics and observed that latency persisted. In that case, cache was poisoned because a code change in that deployment. The system would required some time to return back to normal after rolling back the code. Had the on-call engineer identified that root cause, he could have confidently awaited for the system latency to decrease to normal.

### 5.2.6 Handing off

The action of passing responsibility for the incident response to another. We observed this action in the following cases: (1) different ownership, (2) not having the technical knowledge, (3) on-call rotation, and (4) distribution of investigation to many engineers.

1. Different ownership: entrust incident response to an engineer or team with proper credentials to investigate and solve issue. In P4.I1 and P4.I2, the initial investigating engineers identified that problem was coming from another system. The engineer handed off the incident to the owner of that faulty system. Same case with P11.I2, where the call leader, after solving a problem, called other on-call engineers to verify that their systems were working properly.
2. Technical knowledge: entrust incident response to an engineer or team with better technical knowledge. In both P1.I1 and P1.I2, a supervisor without technical knowledge handed off the incident to an engineer. Also, In P5.I2, the initial investigating engineer was not able to understand and solve the problem, so a senior engineer took over the incident response. Same with P10.I1, where an incident was handed off to a group of experts who solved the incident.

3. On-call rotation: entrust incident response to the next on-call engineer or team, in cases when there was a on-call team for every period of the day. In P7.I1, the SRE team handed off the incident to the next on-call team from a different time zone. In P9.I1, one engineer detected a problem and handed off the incident to the on-call engineer responsible for a system.

4. Distribute investigation: entrust a part of the incident response and investigation to an engineer or team. Common in situations where all hands must be on deck, a senior engineer or manager may coordinate such effort. In P8.I2, a senior engineer handed off, for every engineer, responsibility over investigation of a part of their systems.

Similar to diagnosing, a hand off can be guided by the amount of information a support engineer has at her disposal. A hand off can be guided by protocols and rules. A support engineer can also hand off an incident to another engineer with ownership and authority to investigate and act upon a potential faulty component. In the cases where more than one engineer has authority to investigate a component, a support engineer with lack of technical knowledge on an issue may hand off the incident to a more experienced engineer. Incidents P1.I1, P1.I2 and P7.I1 have examples of hand offs guided by protocols or rules. Incidents P4.I1, P4.I2, P9.I1 and P11.I2 have examples of hand offs to engineers with ownership to a potential faulty component. Incidents P5.I2, P8.I2 and P10.I1 have examples of hand offs to more capable and experienced engineers.

We observed that an incident is only going to be mitigated when it is handed off to the right engineer or team, be it because of technical knowledge or credentials. In an ideal
scenario, the notification of an incident always should be sent directly to an engineer or team capable of solving the problem.

5.3 Key Observations

Our study is focused on improving the incident response by providing important theoretical information that can be used to improve monitoring tools, help with the design of training for incident response and create more robust systems. In this study we focus our validation on the improvement of monitoring tools. One aspect that can improve a monitoring tool is the capacity to potentially decrease time to mitigation. Based on the analysis of the incidents that were used to create our ontology and model, we will provide some key observations that may lead to a decrease in time to mitigation. In key observation 1, we discuss the detection of incidents. In key observation 2, we discuss triage and the hand over process. In key observation 3, we discuss how to provide information needed for an opportunistic search. Finally, in key observation 4 we discuss resource limitation incidents and a potential way to prevent them.

5.3.1 Key observation 1: automatic detection will potentially change late detection to on-time detection. Also, certain symptoms types may be harder to catch.

One way to decrease time for mitigation is to turn a late detection into an on-time detection, and an on-time detection into an early detection. In our data, every manual detection is a late detection, unless the first user or an internal team reports the error immediately after the impact. Even in these cases, the delay for an engineer to receive a notification could be avoided if an automatic tool was present. The only case where a manual detection was an on-time detection was when the team responsible for a system was also its user. Such is the case of P12.I1 and P12.I2. On the other hand, every automatic detection happened on-time.
The monitoring tools were tuned to detect issues that would impact the mission of a system. Knowing this, we analyzed our data to understand what is hard to monitor and detect.

Of our 11 automatic detection cases, 9 captured a generic symptom while 2 captured a specific symptom, a sign that it may be easier to detect generic symptoms using automatic tools. This is implied by P4, when he noted:

We have to monitor data quality, which is a lot harder than just monitoring a CPU and memory, because those are static values. [...] so now we are trying to figure out how to monitor data quality... So that’s kind of the current problem generation we are dealing with.

On the other hand, we observe more detection cases of specific symptoms when detection is manual. Of our 11 manual detection cases, 4 started with a generic symptom and 7 started with a specific symptom. Of these 4 generic symptoms, 3 could easily be detected by a monitoring tool, which would potentially decrease time to mitigation. The remaining 1 generic symptom was a drop in operations, something that in normal circumstances would not be considered a symptom of a system failure.

Basic resources, like disk space, CPU, or memory, are necessary for most system types, and the lack of these resources can indicate an incident. They can easily be monitored. Monitoring tools from our data also often monitored latency and outages, two important metrics for the mission of a system. Specific symptoms, on the other hand, often require monitoring of specific metrics. In P6.II and P9.II, the only cases where a monitoring tool captured a specific symptom, specific types of errors were purposely monitored and captured. In the 7 cases where a specific symptom was captured manually, 6 of them (P1.II, P3.II, P4.II, P6.II, P10.II, and P11.II) probably could not be captured by monitoring generic metrics because they would not generate a generic symptom like a latency or an outage. For instance, in P1.II, a website was showing a different design than expected. In P11.II, some content was unavailable to users. They were not captured because they were not predicted. Automatic detection tools in our data are manually predicted and deployed. These
unpredictable issues generated specific symptoms and would demand specific monitoring measures to be detected. For instance, P1.I1 could be detected by a script to verify if a design update worked as expected. Or P11.I1 could be detected if a script would try to find every content from cache as a user. Only P2.I2 would be captured if the latency of one specific operation was monitored. We believe that it may be unfeasible to expect that a software engineer will predict and manually deploy a monitoring measure for every possible error. One way to improve this is to automatically detect incidents without the need of human prediction.

Current research can potentially help with such issues. One example is the creation of a tool that can automatically detect anomalies on invariants, meaning properties of a system [19]. Hangal and Lam claim that these tools are important to find corner cases during debugging and program evolution [24]. Although these detection tools are capable of helping a software engineer during coding or a support engineer during investigation, analysis of our data suggests that it would be as important to develop and use these tools to also detect issues and anomalies in the system automatically, including types of anomalies that engineers may not have determined in advance that needed to be monitored.

5.3.2 Key observation 2: elimination or improvement of hand offs can decrease the incident response time

Hand offs can result in increased mitigation time. Out of our 22 incidents, 10 had the presence of a hand off and 12 had not. In all 12 incidents without a hand off, the receiving engineer had the authority to mitigate the issue. The engineer also had knowledge or no other choice but to learn what was needed to respond to the incident. In our 10 incidents with a hand off, 3 were received by someone without authority to act or mitigate a faulty component (P4.I1, P4.I2, P9.I1), 5 were received by someone with authority but no knowledge or capacity to act and solve a problem (P1.I1, P1.I2, P5.I2, P7.I1, P10.I1), and the remaining 2 demanded multiple mitigation steps from different engineers (P8.I2 and P11.I2). This means that a
hand off may be necessary when the response must be done in more than one step by more than one engineer. Also, to improve the hand off process, the incident must be sent to a support engineer with knowledge and ownership. Given the time limitations of this study, we will focus on improving cases where the incident is sent to a support engineer without access or authority to make changes on a faulty component.

Analysis of cases P4.I1, P4.I2 and P9.I1 shows that all engineering teams had dashboards and monitoring tools to detect incidents. Further analysis reveals that the reason the incident was sent to the wrong team is because the team receiving the incident had access to the symptoms but no access to metrics related to the root cause. On the other hand, the team with a defective system do not had access to symptoms. This division between symptoms and system related metrics hinder the investigation and also the hand off itself. For instance, in P4.I2, P4’s team discovered that an issue was not related to their systems. With limited information on where the problem could be, the only choice P4’s team had was to hand off the incident to the team providing their data. This pattern continued until the incident was handed off to a team responsible for a faulty component. More data was necessary in order for P4’s team to hand off the incident to the right team. To solve this problem, we propose a design improvement in chapter 6.

Some research on software engineering can be useful for the hand over process. For instance, Anvik and Murphy created a recommendation system to help support engineers based on bug reports, changing the role of gathering information during triage to recommend engineers the proper team capable of solving the issue [3]. Applying bug triage to incident triage, Chen et al. tested 6 different automatic bug triage techniques for automatic incident triage on online service systems, finding out that these techniques can achieve an accuracy of 32% to 71% depending of which of the 6 techniques were used [14]. They noted that, while bugs are treated individually, incidents are correlated to time and events, and techniques that consider correlations should improve the accuracy of automatic incident triage. Our proposed design improvement at chapter 6 also seeks to build upon these suggestion.
5.3.3 Key observation 3: to follow an opportunistic search guided by symptoms, engineers need easily available information

While an engineer using a systematic search strategy will walk the chain of events until a root cause is found, an opportunistic search can help one engineer save time by getting a ‘shortcut’ and become closer to a root cause. To follow an opportunistic search and investigate closer to a root cause, support engineers need knowledge, experience or information.

We observed the presence of an opportunistic search guided by symptoms in 7 out of our 22 incidents: P1.I3, P2.I1, P5.I1, P5.I2, P8.I1, and P11.I2. Although every participant in this study had at least some sort of supporting tool at his or her disposal, we are unable to evaluate and compare the supporting tools of each participant. Nonetheless, we can observe some attributes of incidents without the presence of a guided search:

1. A support engineer does not need symptoms to begin an investigation when the initial detected symptom of an incident is specific. None of the 8 incidents starting with the detection of a specific symptom used an opportunistic search guided by symptoms, meaning that the engineer was able to begin an investigation without the need for further information. Even in these cases our analysis suggests that a tool providing easily available information would still be useful for the support engineer.

2. Symptoms and system information may not be readily available to the support engineer. The common investigation action during diagnosing are engineers seeking information from the system. In our 15 incidents without the presence of an opportunistic search guided by symptoms, the support engineer had at least some sort of support tool to provide some type of information, be it rudimentary or advanced. The fact that the support engineers had to delve and search for some sort of information rather than use the support tool is an indicative that such need for such information was not predicted and deployed.
The lack of some type of information readily available may hinder the effectiveness of a support tool. For instance, Lou et al. created a monitoring tool based on software analytics with the intent of helping support engineers find faulty components during their investigation [30]. Their tool helped support engineers find a proximate cause of an incident in 76% of cases. Lou et al. observed that the tool failed to help support engineers find the cause in 24% of the cases because it was not retrieving logs or data related to an incident. As we noted, support engineers were not able to use their available tools in 15 cases because the desired information was not being provided by a tool. It is unrealistic to expect that an engineer will predict every possible error. Also, it is unrealistic to expect a supporting tool to show every possible information and metric. Knowing these points, we propose a small design change on monitoring tools to address these issues.

5.3.4 Key observation 4: Stress and shock as types of incidents, and potential incident prevention methods

As part of this study, we attempted to collect as much data as possible related to how one incident happened to be, rather than focusing solely on the response. Analysis of how one incident happened helped us to identify important concepts about incident types: limit, stress and shock. The latter two terms are borrowed from mechanical engineering, and are present in different engineering areas like civil engineering. Shock is defined as a sudden and severe force that usually causes a displacement on a mechanical system [25]. Stress is defined as a force excerpted over a point, even when external forces are absent [4]. Stress and shock can provoke temporal or permanent deformations, called strains. To prevent incidents related to stress and shock, engineers perform structural calculations to ensure that a system will work under certain conditions. For instance, while designing a bridge, an engineer will perform a structural analysis, reinforcing confidence that a bridge will hold given certain conditions. This calculus will consider the structure, materials, and even consider that different components and regions of a bridge will receive each different levels of stress.
There are also calculations of external and dynamic factors, like wind. These procedures and studies were fuelled by case studies of failures, as noted by Dellate and Spurrier [18, 42].

These same terms can be used in software engineering and explain types of issues. Nygard uses the terms impulse (as a fast shock) and stress to define types of incidents [33]. For instance, he used a tweet from a celebrity about a website or a flash sale as impulse examples. He also used the small capacity on one component of a system as stress, noting that it can affect other components connected to this component with limited capacity. In a similar way, we can observe these characteristics in our incidents.

Our data confirms 10 incidents that are related to stress and shock. Of these incidents, 9 are the result of a lack of some sort of calculation of resources. They are P1.I2, P2.I2, P4.I2, P5.I1, P6.I1, P6.I3, P8.I1, P11.I1, and P12.I1. The only exception is P1.I3, a stress and shock incident resulting of a denial of service attack. We were unable to identify if stress was related to 6 of our incidents, given the lack of detail on how these incidents happened. These incidents are P1.I1, P2.I1, P3.I1, P6.I2, P11.I2, and P12.I2. All 6 remaining incidents are the result of some sort of dependency or system property change unbeknownst to the engineer. They are P4.I1, P5.I2, P7.I1, P8.I2, P9.I1, and P10.I1.

Current ongoing research can help with resource management issues. For instance, research on resource monitoring can help engineers make decisions before a limit is reached [5, 45], while research on auto-scaling resources can remove the responsibility of an engineer to manually detect when a resource should be scaled [8, 13, 41, 51]. Engineering resources management is, according to Jennings and Stadler, one of the biggest challenges in distributed systems and much ongoing research in being done [26]. Despite these tools and ongoing research, analysis of our data shows that stress incidents will continue to happen because the engineer designing the system needs to be aware of all resource limitations of their systems and deploy managing tools accordingly, be it a resource monitoring or auto-scale. It may be implausible to expect an engineer to predict every existing limit of their systems. For instance, every stress incident besides P1.I3 could be easily prevented if the engineer deployed
a resource monitoring tool or an auto-scaling tool. The companies and engineers in these cases had the technology and capacity to deploy such tools. However, they did not deploy such measures before the incident because they were not explicitly aware of the limitations of their systems. In fact, with exception of P1.I2 and P1.I3, these stress incidents were the first of their kind, meaning that after they happened, tools or changes were deployed to prevent or detect such incidents. The development and deployment of a tool capable of identifying every resource limitation automatically within a system would inform engineers and help them prevent stress incidents.
Chapter 6

Validation

As stated before, we will validate the usefulness of our ontology and model by using our analytical results to propose a better design for tools used during incidents. Specifically, based on some key observations discussed in sections 5.3.1, 5.3.2 and 5.3.3, we will describe one improvement for a dashboard tool.

Figure 6.1: A common dashboard tool.

Dashboards provide accessible and clear information for support engineers during incidents. Engineers will chose events and metrics from the system and use those to create dashboards that will support engineers during a range of incidents. The engineer will not include all metrics or every existing event type, given that an overload of information can make an engineer miss one important metric related to an incident [50]. Faced with too much information, Bawdney et. al points out that a possible solution is an agent capable of filtering information for their user [7].
Our improvement suggestion is to create a tool that will provide dynamic metrics temporally related to an incident when an alarm is fired. The monitoring system will have a correlation agent that will read logs and events. When activated by an alarm, it will provide metrics and events correlated to an incident. The objective is that the monitoring system will help the engineer identity possible faulty components of the system.

For example, during incident P5.I2, a monitoring system detected high latency. The proposed tool would receive this alarm and send to the support engineer a dashboard with possible related metrics, like latency, increased back-end requests, decreased cache readings, and events like recent or related deployments. We claim that this would help even inexperienced engineers find the possible components to be investigated. This would also help engineers ignore misleading signals. For instance, in P7.I1, ongoing errors from a system not related to the incident confounded the investigating team during their investigation.
A correlation engine would be able to read these constant errors and identify them as not related to an incident.

The proposed tool should also be able to help an engineer hand off the incident when needed. To achieve this, the correlation agent must read logs from the whole ecosystem of systems, not being limited by one system. For instance, in P4.I2, the first investigating team had metrics and dashboards related to their systems. Unfortunately, the incident was related to a system they did not own and was not included in their dashboards. The correlation engine would be able to correlate the sales drop to the 503 errors from load balancers assuming access to logs across the system were granted.

The correlation engine would provide generic information and specific information. It would also help the engineer hand off the incident to the right team fast. Generated metrics from this agent would provide a shortcut, approximating support engineers to possible root causes.

This tool would address an issue observed in section 5.3.1, helping automatic tools provide specific symptoms in addition to generic symptoms. It would address a concern in section 5.3.2, quickly pointing out potential faulty components by correlating metrics from different parts of a system. This fast information would help a support engineer know if the incident must be handed off, and who should receive the incident. It would also address section 5.3.3, providing specific information that would help the engineer follow an opportunistic search strategy, providing a shortcut during the investigation to the support engineer.
Chapter 7

Discussion

In this study, we were able to create an ontology of incidents and a model of the incident response. We also described some analytical observations and validated the usefulness of our analytical results by proposing an improvement for tools used by support engineers. In this section we discuss a series of limitations related to how our data was collected. We also discuss some future work that will greatly improve incident response and the prevention of incidents.

7.1 Limitations

As stated in section 4.2, given all possible types of incidents that may exist and our time frame, our resulting ontology and model of incident response is not 100% comprehensive. Also, an ontology may demand continuous work and change, meaning that their terms must be updated to keep up with evolving topics and business dynamics [43].

Another limitation of this work is the data collection method. The use of interviews was an effective way to collect our desired data. We disregarded the use of surveys, since it would not produce sufficiently detailed data for an exploratory study of this nature. Another option would be shadowing and field studies. We understand that some data can only be obtained by shadowing and by field studies. For instance, in one study an observer discovered that a mistake during a troubleshooting activity was due to the lack of knowledge of the support engineer [6]. This is a type of error that may not be identifiable using interviews. Nonetheless, such methods are costly and more time consuming, as one reported case of the
previous cited study that took two weeks. The use of shadowing and a field study demands more time to observe the same number of events that we obtained through interviews. Also, it is hard to do theoretical sampling using such methods, since incidents are unexpected events. This limitation becomes clear as a researcher looks for a rare and odd incident that may never occur during a field study.

The use of interviews also implies certain limitations here discussed. We first recognize a memory limitation. For instance, an interviewee may not remember all steps of an incident, limiting the details available. A second limitation is about limitation to one’s own experiences. For instance, support engineers may report the details of their actions and tasks, but they face some difficulty giving details of the actions of other professionals during the incident response. Such information is not as reliable as personal experience. A third limitation may come from bias from interviewees, as they can exaggerate some facts, attribute positive outcome from their actions and negative from others, or even have an incomplete comprehension of the actions done, misleading the interviewer. To minimize these problems, we asked our interviewees to share only their own experiences, and we asked specifically for recent incidents. Before the interview, participants were told that they would be asked to share four to five experiences dealing with incidents, avoiding taking them by surprise and helping them remember the incidents before the interview. Lastly, we recognize that by asking participants to share incidents whose memory about details are precise, we also captured outstanding cases, since these critical cases leave a longer impact on someone’s mind. While this may supply us with critical failures or corner cases important for a theoretical sampling, we also may have missed common and ordinary day to day incidents.

7.2 Future Work

By observing incidents in our data, how they happened and our observations in section 5.3.4, we believe that the improvement of the current research on resources management will help prevent stress related incidents. Also, as noted in some other engineering areas, engineers
will follow methodologies, perform calculations and follow norms to ensure that an object or building will hold. We believe that the development of tools capable of automatically predicting resource limitations and potential failure from systems will help engineers prevent incidents and become better prepared for future incidents.

Furthermore, future research on incident response can improve the identified concepts in this study. As noted, this was not an exhaustive study of incidents. More incident cases would greatly improve the ontology of incidents and the model of incident response, increasing its completeness based on data outside this study. An open and well detailed database of incidents would also help the research community. One example of such database is the Veris Community Database\textsuperscript{1} for security incidents. Another list with links to post-mortem publications at varied websites is a published list of post-mortems \textsuperscript{2}, with reports following different formats and with different granularity levels. To the best of our knowledge, no open, consistent, and detailed database of incidents with incident responses is available to the research community. We claim that the creation and continuous expansion of such a database would foster research about incidents and the incident response.

7.3 Summary

This exploratory study based on 22 incidents was used to identify the core concepts of incidents. It resulted in a theoretical basis model for the ongoing study of the incident response, namely an ontology of incidents and a model of the incident response. We provided key observations from incidents based on our ontology and model of incident response. This model and ontology will improve monitoring tools, incident response training and the prevention of incidents in software engineering. We validated our results by using the ontology, model and our analytical results of our incidents to propose improvements on monitoring tools for incident response.

\textsuperscript{1}\url{http://veriscommunity.net/verdb.html}
\textsuperscript{2}\url{https://github.com/danluu/post-mortems}
7.3.1 Acknowledgments

The author is grateful to the participants who willingly provided data for this study. The author also acknowledges the help and support provided by the department of computer science at BYU.
References


59


