An Empirical Study on Quality Issues of Production Big Data Platform

Hucheng Zhou*, Jian-Guang Lou*, Hongyu Zhang*, Haibo Lin†, Haoxiang Lin*, Tingting Qin*

*Microsoft Research, Beijing, China

Email: {huzho,jlou,honzhang,haoxlin,tiqint}@microsoft.com

†Microsoft, Beijing, China

Email: haibolin@microsoft.com

***Abstract*— Big Data computing platform has evolved to be a multi-tenant service. The service quality matters because system failure or performance slowdown could adversely affect business and user experience. To date, there is few study in literature on service quality issues of production Big Data computing platform. In this paper, we present an empirical study on the service quality issues of Microsoft ProductA, which is a company-wide multi-tenant Big Data computing platform, serving thousands of customers from hundreds of teams. ProductA has a well-defined escalation process (i.e., incident management process), which helps customers report service quality issues on 24/7 basis. This paper investigates the common symptom, causes and mitigation of service quality issues in Big Data platform. We conduct a comprehensive empirical study on 210 real service quality issues of ProductA. Our major findings include (1) 21.0% of escalations are caused by hardware faults; (2) 36.2% are caused by system side defects; (3) 37.2% are due to customer side faults. We also studied the general diagnosis process and the commonly adopted mitigation solutions. Our study results provide valuable guidance on improving existing development and maintenance practice of production Big Data platform, and motivate tool support.**

***Index Terms*- empirical study, Big Data computing, quality issues, escalations, fault tolerance.**

I. INTRODUCTION

Big Data computing infrastructures, including distributed storage systems (GFS [12], HDFS [4]) and distributed data-parallel execution engines (MapReduce [8], Hadoop [3] and Dryad [16]), are prevalently used for storing and processing web-scale data. ProductA is a distributed Big Data computing platform that is used in Microsoft for storing and analyzing massive amounts of data. Many Microsoft product teams are using ProductA for tasks such as web-scale data mining, ranking search results, and business intelligence. ProductA has also evolved to be a company-wide large ecosystem, where customers share the storage and processing clusters with hundreds of thousands commodity servers. There are tens of thousands of jobs executed by ProductA per day, with different workloads and scenarios. The scale is still in fast increasing.

ProductA is designed to be fault tolerant: jobs that failed are retried when needed. Although a variety of hardware and software faults are tolerated, still some jobs fail or suffer performance slowdown, which adversely affects the business and the user experience. When jobs fail or slow down, customers expect the ProductA support team to investigate and resolve issues on 24×7 basis. It is challenging to guarantee service quality for all users with different service level agreements (SLAs). ProductA has a well-defined incident management process that helps customers deal with the live site issues as they come in. When customers encounter major problems, they can escalate the issues via email. These important customer issue reports, which are called $escalations$ internally, contain the descriptions of the issues experienced by customers, the business impact, and the relevant reproduction and troubleshooting details. The escalations are usually stored in an internal tracking system. Designated Responsible Individuals (DRIs) of the ProductA support team are responsible for tracking and resolving the escalations. DRIs prioritize all the escalations based on their business impact and severity. They are also responsible for quick mitigation of the issues and follow-up postmortem analysis of the root-causes.

Over the years, many empirical studies have been performed to understand the characteristics of quality issues of conventional software systems (e.g., [24], [20], [7], [10], [2], [29]). However, to our best knowledge, few empirical study was carried out for large-scale Big Data computing platforms, which are a modern form of software system. We believe that it is important to understand the characteristics of the quality issues of Big Data computing platforms, so that we could apply the findings to guide rational system design and operation.

In this paper, we present our study on the quality issues of ProductA[1], in order to achieve a better understanding of the quality issues associated with a Big Data computing platform. We manually examine 210 randomly sampled escalations for ProductA. Our study aims to address the following three questions:

1) RQ1: What are the common symptoms of quality issues in Big Data computing platform?
2) RQ2: What are the common faults that hurt service quality of Big Data computing platform?
3) RQ3: What are the common mitigation, and what are helpful in making fast mitigation decisions?

Answers to these questions are generally useful for both development engineers and maintenance engineers, to improve their daily activities including system operation, infrastructure design, and software development. The study also motivates automatic diagnosis tools to reduce the manual maintenance efforts.

The rest of the paper is organized as follows. Section II provides a brief primer on ProductA. Section III describes the design of our empirical study. Section IV, Section V,

[1]Please note that the survey was conducted in 2012. The results presented in this paper may not necessarily reflect the quality status of the current version of ProductA.

and Section VI present the common symptoms, causes, and mitigations of the escalations, respectively. We discuss the issues and lessons learned from this study in Section VII. Section VIII introduces the related work, and Section IX concludes this paper.

II. Background: The Big Data Computing Platform

ProductA includes a distributed file system like GFS and a distributed data-parallel execution engine like MapReduce. In ProductA, a program is written in a language called LanguageB, which is a hybrid language that consists of declarative SQL-like queries and imperative user-defined functions. In ProductA, a data file stored in ProductA is called a *stream*, which consists of data blocks called *extents*. A data-parallel program is called a *job*, and an execution unit is called a *vertex*.

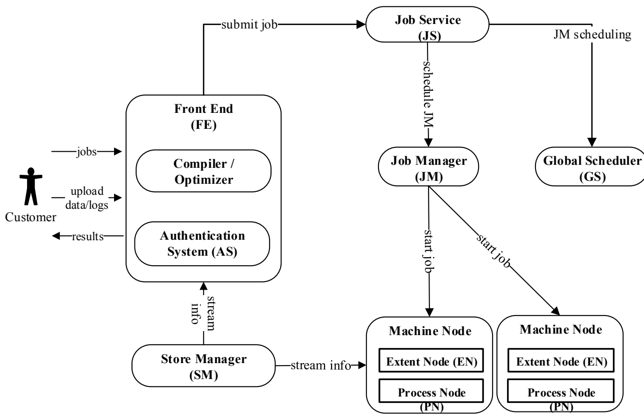

Fig. 1. The architecture and workflow of ProductA.

Figure 1 depicts the brief system framework of ProductA, which includes the following components:

- A front end web service (*FE*) for bridging between customer requests and the system,
- A *compiler* and *optimizer* for job compilation and optimization;
- An authentication system (*AS*) for authenticating customer requests;
- A store manager (*SM*) for managing distributed file system metadata (similar to the *master* in GFS);
- A job service (*JS*) for job queuing and scheduling as well as resource capacity management;
- A global scheduler (*GS*) for maintaining machine health states in a data center;
- A job manger (*JM*) for coordinating the distributed execution of job vertices;
- Extent nodes (*EN*) for storing data extents as local files (like what in GFS' *ChunkServer*);
- Processing nodes (*PN*) for executing the vertices and communicating with the job manager via heart-beating.

Each job has a job manager, which is actually a special vertex. EN and PN services are deployed in all servers in the data center.

In ProductA, customer requests are dispatched to one FE server and authenticated by AS. If authentication succeeds, FE queries the location of each data extent and then the customer can directly access the ENs through a ProductA client library.

Jobs in ProductA could fail or have poor performance. For ease of diagnosis, the system records important event logs and stores the daily collected telemetry data as a ProductA stream, which can be later used for job monitoring and postmortem analysis. All computation (JM, PN) and storage (SM and EN) components have such event tracing mechanism. To improve user experience, ProductA also provides multiple user interfaces such as a web-based UI and a Visual Studio plugin.

Considering the increasing scale of ProductA, it is challenging to guarantee service quality for all customers with different SLAs. ProductA provides incident reporting mechanism to customers, and the incidents are expected to be mitigated quickly by the support team (aka *Designated Responsible Individuals (DRIs)*).

III. Study Design

In our empirical study, we randomly collected 210 service quality issues (escalations). These issues were discussed in 2,196 emails and 188 incident tracking records. Additionally, we also collected the corresponding job information such as initial input data, source scripts, execution plan, and runtime statistics to understand more about a specific escalation. We included both emails and incident records because they are complementary to each other. Emails describe the whole life cycle of an escalation, including its submission, the first designated responsible individual (DRI) engagement, and the final mitigation. Incident records are much more structured and formal.

The study includes two parts: (1) manually reading the content of escalation emails and incident records, identifying the detailed diagnosis process and mitigation actions, and reasoning the possible root causes. (2) automatically extracting the metadata information from the collected escalation database and calculating the metrics on DRI involvement, including the *time to engagement (TTE)* and the *time to mitigation (TTM)*.

In our study, we first classify the escalations into five categories from the symptom point of view, including job failure, job slowdown, connection failure, service unavailable and wrong result. We further classify the common causes into three categories, including hardware faults, system side faults and customer side faults. Lastly, the mitigation solutions are also classified from the resolution point of view.

IV. What Are the Common Symptoms?

We first conduct a study on the common symptoms of quality issues of Big Data computing platform, by manually reading the escalations customers submitted. From the symptom point of view, the escalations can be classified into five categories, including connection failure, job slowdown, job failure, wrong result, and service unavailable. Table I gives the classification statistics, which show that a majority of escalations are exhibited as job failures (45.2%) and job slowdown (27.1%).

TABLE I. CLASSIFICATION FROM ESCALATION SYMPTOM POINT OF VIEW.

Category	Number	Ratio
Connection Failure	18	8.6%
Job Slowdown	57	27.1%
Job Failure	95	45.2%
Wrong Result	18	8.6%
Service Unavailable	12	5.7%

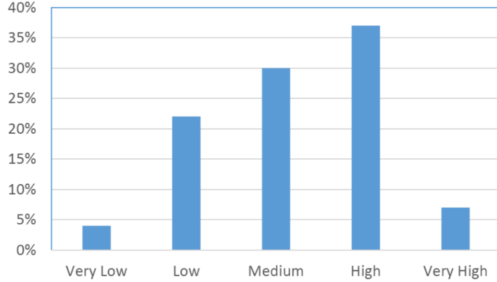

Fig. 2. The distribution of escalation severity

We further study the severity of the escalations. When escalating the service quality issue to ProductA team, customer is obligated to state the severity level, which helps determine the DRI engagement priority and the required mitigating time. There are five severity levels, ranging from Very High, High, Medium, Low, to Very Low. Escalations at the Very High level must be engaged within 15 minutes and mitigated within 1 hour; while the Very Low severity ones could be engaged with the lowest priority and mitigated in one release cycle, if they are worth of fixing. The severity level is assessed based on both business significance and impact of the issue. Figure 2 depicts the severity distribution among 210 escalations we studied. Only 7% of escalations are at very high level, 37% are at high level, 30% are at medium level, 22% are at low level, and about 4% are at very low level.

V. WHAT ARE THE COMMON CAUSES?

We have identified the causes of the 210 escalations manually, and classified them into three categories, including hardware faults, system side faults, and customer side faults. Table II depicts the detailed categories. For most of the escalations (94.4%), their causes can be successfully classified. For the rest of the 5.6% (12) of the escalations, their causes are unknown due to the missing evidence. We describe each cause category in details in the following sections.

A. Escalations Due to Hardware Faults

Modern data centers are built with commodity machines that have relatively high probability of failure. Our study shows

TABLE II. CLASSIFICATION OF ESCALATIONS CAUSES.

Category	Sub-Category	No.	Ratio
Hardware fault	Subtotal	44	21.0%
System side fault	System code defect	44	21.0%
	Design limitation	20	9.5%
	Operation fault	12	5.7%
	Subtotal	76	36.2%
Customer side fault	Code defect	10	4.8%
	Operation fault	21	10.0%
	Misuse	47	22.4%
	Subtotal	78	37.2%

TABLE III. CLASSIFICATION OF HARDWARE FAULTS.

Category	Sub-Category	No.	Ratio
Hardware issue	Machine unhealthy or outage	21	10.0%
	Power off	5	2.4%
	Network device fault	9	4.3%
	Overheating	2	1.0%
	Bit flipping	3	1.4%
	Time drifting	4	1.9%
	Subtotal	44	21.0%

TABLE IV. CLASSIFICATION OF SYSTEM SIDE FAULTS.

Category	Sub-Category	No.	Ratio
System code defect	Regression	14	6.7%
	Component related	16	7.6%
	Service outage	12	5.7%
	Memory issue	2	1.0%
	Subtotal	44	21.0%
Design limitation	Extreme situation	6	2.9%
	Resource contention	8	3.8%
	Resource overloaded	6	2.9%
	Subtotal	20	9.5%
Operation fault	Subtotal	12	5.7%

that hardware fault is one of the most common causes that lead to escalations. We further divide the hardware faults into several sub-categories (Table III), including machine outages, power off, network device fault, time drifting, and overheating. Some sub-categories were also observed and reported by existing studies on hardware faults [22], [23]. Besides, our study reveals one unusual type of hardware faults, namely bit flipping. Bit flipping is an unintentional state switch from 0 to 1, or vice versa. There are 1.4% (3) cases where bit flipping caused corrupted data. Bit flipping in memory happened when a stream was generated and before the stream is stored into persistent storage, but it was detected when the stream was read and parsed by the later job. Re-executing the same job could mitigate this problem.

B. System Side Faults

In this section, we describe our study on system side faults, which are further classified into system code defect, design limitation, and operation fault.

1) System Code Defect

It is unsurprising that code defects in ProductA account for 21.0% (44) of the escalations, because escalations are sent out only if customers consider them as system-related faults.

Regression. Only a few of (about 6.7% (14)) escalations are due to system regression. This is because the roll-out policy adopted by ProductA. The service-oriented architecture advocates agile and independent development in each sub-service teams. Currently new features are rolled out incrementally: ProductA team first selects part of the customer jobs from specific feature team for partial release; and gradually enlarges the scope until fully deployed. Historical versions of system are maintained, thus customer can choose which version to use by configuring the job submission parameter. Once regression happened, job resubmission with older runtime could mitigate it quickly.

Component related. 7.6% (16) of escalations are caused by code defects in different system components, including FE, JS, JM,SM, EN and optimizer. Each component results in about three escalations. These defects are mostly trivial code defects. Only two bugs in LanguageB query optimizer are related to Big Data computing. One example is that, the optimizer spent

more than 25 minutes without any update and the job thereby failed to submit. This is because the job was large and the optimizer failed to enumerates all possible execution plans to select the one with minimum cost. Another example is that the optimizer tried to repartition the large execution graph, but it finally failed.

Service outage. 5.7% (12) of escalations are caused by service outages. Among them, 83.3% (10/12) happened in FE, 8.3% (1/12) in EN and 8.3% (1/12) in compiler. The FE outages happened at only three time points. The root causes are hard to diagnose, and the corresponding mitigation is just to restart the service.

Memory issue. Our study shows that 1.0% (2) of escalations are related to memory issues. These escalations could be avoided by improving the system design.

2) System Design Limitation

In addition to code defects, we find out that some customer issues are related to system design limitations. 9.5% (20) of escalations could be avoided by a better design.

Extreme situation. Six escalations are exposed only in extreme situation, where system design limitations surface. For example, there is a job that contains a join between a huge file (with tens of tera bytes) and a smaller file (with tens of mega bytes), and the customer splits both files into 8,000 partitions in order to gain more parallelism. All partitions of the files are located at the same single machine, thereby 8,000 vertices in join stage read that machine (even 3,000+ simultaneous read), which results in overloaded machines and failed vertices. Things could become better if the optimizer applies the broadcast join or the job manager throttles the vertices scheduling to a single machine. Another interesting example is a job failure that resulted from repeatedly data access error. Data de-replicating process aims for saving space. There are usually three replicas when data stream first gets created, and it turns into two replicas by de-replicating one. If a vertex is accessing one data replica that is under de-replicating, the vertex would repeatedly failed. By introducing inter-awareness between job scheduling in job manager and the de-replicating process in SM, such cases can be avoided.

Resource contention and overloaded. We find that resource contention and overload caused eight and six escalations, respectively. These escalations can be avoided by adopting a proper design. One escalation was due to workload imbalance with huge spike in traffic. There was also a job slowdown happened because some of the vertices were affected by other activities (such as data downloading request from other customers). A contention-aware scheduling, or a performance isolation design, could largely alleviate these issues.

Finding 1: 21.0% (44) of escalations are caused by system code defects, 5.7% (12) are due to operational faults, and 9.5% (20) are due to design limitations.
Implication: While it is not easy to provide multi-tenancy Big Data computing service, there is still room for better design, implementation, and operation. Testing in a broader scope, especially online testing in real production, would be helpful to expose the system faults as early as possible.

TABLE V. CLASSIFICATION OF CUSTOMER SIDE FAULTS.

Category	Sub-Category	No.	Ratio
Code defect	Buggy and non-optimized code	8	3.8%
	Inhibitive programming style	2	1.0%
	Subtotal	10	4.8%
Operation Fault	Subtotal	21	10.0%
Misuse	Incorrect client configuration	15	7.1%
	Improper job parameters	9	4.3%
	Improper system assumptions	23	10.9%
	Subtotal	47	22.4%

3) System Operation Fault

System operation faults include deployment faults, incomplete provision, and library version mismatch, etc. One example is that the deployment of a service component restarted thousands of extent processes (ENs), causing intermediate data loss. Another interesting example is that an operator manually updated one service component at the live site, which corrupted the system caches. These warn us that we should be cautious and pay more attention to system management operations. Better process management and guidelines are helpful here.

C. Customer Side Faults

It is surprising that 37.2% (78) of escalations are still caused by customer side faults, considering that escalations are only reported if customers think that the problems are not at their side. The statistics are shown in Table V. The customer side faults include 4.8% (10) code defects, 10.0% (21) operation faults, and 22.4% (47) misuses.

1) Customer Code Defect

Customer code defects are not obvious and hard to detect, which include buggy code and inhibitive programming style.

Buggy code. This type of defects includes defects in third-party library and misunderstanding of advanced programming features. Some job slowdowns are due to data skew, where groups with some keys are much bigger than the others, thus the corresponding vertices become the "outliers" [1]. Code defects also happen to third-party libraries, which are invisible and hard to detect.

The advanced language features provided by LanguageB could also introduce defects. Like the *combiner* in Hadoop, LanguageB allows customer to write user-defined recursive reducer, which provides partial aggregation optimization [28]. Instead of sending all mapper data to reducer, it partially aggregates the data on each mapper side, recursively combines the intermediate result, and finalizes the result in reducer side. Taking SQL clause *SUM* as an example, the recursive implementation is to first compute the local sum on each mapper machine, and add them together in reducer. It is computed like a tree, with the output of one vertex being used as the input to the next. Thus it requires the recursive reducer to be associative and commutative, since it would be executed more than once. However, in this example customers did not know such semantic constraints, and obtained the wrong results.

Another more interesting example is about the logical operators. The operators *AND*, *&&*, *OR* and || are used to filter records based on more than one conditions. The expressions in *AND* and *OR* might be evaluated in any order and any number of

```
                                   Script
1 SELECT Market FROM SearchLog
2     WHERE Market != null AND Market.region == "en-us";
3
4 SELECT Market FROM SearchLog
5     WHERE Market != null && Market.region == "en-us";
```

Fig. 3. Semantic difference of *AND* and *&&*.

times; while *&&* and || can only be evaluated in user-specified order with short-circuiting. The reason the evaluation order is not guaranteed in *AND* and *OR* operators is because the LanguageB optimizer can move those predicates where it sees fit in order to improve performance by filtering the useless records as early as possible. It is customers' obligation to use such options judiciously in the places where no dependencies between predicates and reordering is allowed. Incorrect usage would result in wrong output or even job failure. For example, an incorrect use of the *AND* operator is shown in line 2 in Figure 3. The left predicate is a null checking, and the right predicate should be evaluated only if the null checking is satisfied. Otherwise, an invalid memory access failure would be triggered. The correct code is to use *&&*, which is shown in line 5.

Inhibitive programming style. ProductA automatically executes a job in a parallelized and distributing manner, and it discourages customer to write multi-threaded mapper or reducer to gain further parallelism. This would sacrifice the fairness in resource sharing and result in unpredictable execution behavior, which in turn affects the scheduling decision.

Finding 2: 4.8% (10) of escalations are caused by customer code defects, which include buggy and non-optimized code, as well as inhibitive programming style.
Implication: Programs for Big Data computing are hard to diagnose. Tools are needed in order to detect the error-prone patterns and enforce good programming styles in software development.

2) Customer Operation Fault

There are 10.0% (21) of escalations caused by customer side operation faults. There are many kinds of operation faults, such as non-intentional data deletion, wrong data expiration time setting, non-intentional data file renaming, job dependent resource missing, unavailable data, low free storage space, VC switch, and even the zero-sized input data.

3) Misuse

We treat incorrect client configuration and improper submission parameter as misuse. They can be mitigated by resubmission with new configurations or parameters.

Incorrect client configuration. 7.1% (15) of escalations are due to incorrect client configuration, including authorization, proxy configuration, customer library version mismatch, machine-IP address mapping and stream access authorization.

Improper job parameters. 4.3% (9) of escalations are due to improper job submission parameter. For example, the parameter about resource quota is important for job performance. Many escalations can be avoided if the resource quota is increased.

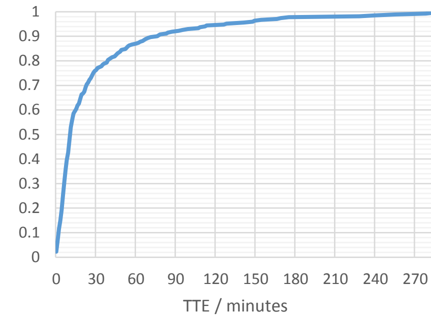

Fig. 4. Cumulative distribution function of TTE.

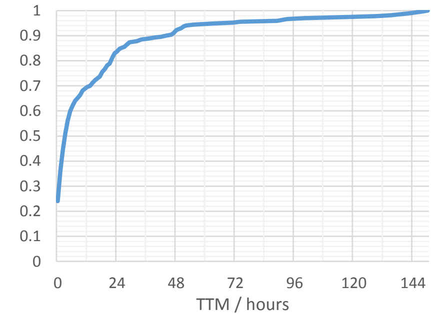

Fig. 5. Cumulative distribution function of TTM.

Improper system assumptions. Systems are always designed with certain assumptions and conventions. The design assumptions should be clearly communicated to and agreed by customers. Our study finds that a large percentage (10.9%) of escalations are due to improper assumptions customers made to ProductA. For example, compiler has an internal limitation on the number of partitions; SM allows limited number of simultaneous stream creations; and ProductA has limitations on the data volume to be sampled. There is one extreme case where customer created more than 1 million streams at a time, which was disallowed by the system. ProductA also kills the jobs that consumes more than 6 GB memory. Without knowing these system assumptions, customers may write programs that fail to be executed.

Finding 3: 10.0% (21) of escalations are caused by customer operation faults, and 22.4% (47) are due to customer misuse.
Implication: Better training and process management are helpful to avoid operational faults and misuse. It would be better if such problems can be detected earlier by an automated tool.

VI. From Escalation to Mitigation

When a service quality issue is escalated to ProductA team, DRIs (Designated Responsible Individuals) are responsible for providing a mitigation solution in time. In this section, we study the mitigation they applied and investigate what kinds of information they used for diagnosis.

A. Statistics on DRI Activity

We use two metrics, Time To Engagement (TTE) and Time To Mitigation (TTM), to evaluate the efficiency of an escalation

handling process. TTE is computed as the time difference between the escalation submission and the first reply email from DRI. TTM is the time difference between the escalation submission and the last email of the escalation email thread. The last email is generally the close of an escalation thus could be used to approximate the TTM. The smaller number is better - small TTE time indicates fast engagement and small TTM time indicates that fast mitigating solution is provided or applied. It is noteworthy that such calculation is very conservative. Some DRI engagements (e.g., those via phone call or instant messaging tools) happened earlier before the first replied emails, and the mitigations often happen before the last replied emails.

The cumulative distribution graphs of TTE and TTM are shown in Figure 4 and Figure 5, respectively. DRI engages with 70% of escalations in less than 30 minutes, 90% of them are processed in less than 90 minutes, and there is no escalation processed in more than 5 hours. Meanwhile, TTM shares the same trend as TTE, but with much longer time to mitigate. About 70% of the escalations are mitigated in less than 12 hours, about 83% are mitigated in less than one day, and the "slowest" mitigation is less than 150 hours.

B. Mitigation Categorization

Compared with bug fixes for traditional programs, the aim of mitigation is to recover the system and resume the impacted jobs as quickly as possible. A mitigation may not need to be a thorough root-cause fix. On the contrary, a work-around solution is needed for most cases. In our study, we find that there are mainly seven categories of mitigation actions that DRIs have adopted. Table VI depicts the detailed categories of mitigation solutions. For only 2.38% (5) of the escalations, their mitigations are unknown due to the lack of explicit descriptions in emails and internal incident records.

Refine customer code, configuration or submission parameter. As mentioned in Section V, about 37.2% escalations are caused by the faults at customer side. From DRIs' perspective, there is no defects in ProductA and no mitigation action is required. However, in current practice, our DRIs help the customers debug the problems by providing the diagnosed causes. DRIs also suggest customers how to refine their code, configurations, and submission parameters.

Resubmit jobs. About 21.0% escalations are caused by hardware faults, and most of them (19.1%) are intermittent faults. Simply resubmitting the failed job could automatically mitigate it, while there is nothing to do for job slowdown. In fact, all machines in our data centers are managed by AutoPilot [15], which detects and recovers software and hardware faults. Once AutoPilot detects a machine being in a faulty or unhealthy state, it automatically de-commits the machine and tries to recover it by restarting or re-imaging the machine. For such cases, simply resubmitting jobs can mitigate the escalations.

De-commit/Restart faulty machines or services. As we have mentioned in Section V, not all kinds of faults could be handled by the fault-tolerance mechanism. In most fault tolerant systems, there are only two health states for a machine or process: Healthy (H) and Faulty (F). However, in reality, a machine (or a process) can be in a "weird" state. A machine (or a process) in a weird state still has heart-beats, but executes abnormally. As an example, a customer reports that their job ran very slowly. After receiving the escalation, DRIs found out that this problem was caused by a disk fault, which took more than 30 minutes to read 2G bytes. Such "weird" machines are likely to become faulty soon [22], DRIs need to label the machine as a bad state and de-commit the machine. Meanwhile, customers still need to resubmit the job. In our study, only 1.4% (3) of escalations are mitigated with extra machine decommission.

Rollback the runtime. Obviously, escalations caused by software regressions can be resolved by rolling back to the previous one. Actually, all latest LanguageB runtime versions are stored in the system. Customers can resubmit their jobs with a parameter indicating the specific runtime version to be used. About 6.7% (14) of escalations are mitigated by runtime rollback.

Resubmit jobs with new parameters to mitigate server side code defects. It is interesting that many failures caused by code defects at the system side can also be mitigated through resubmitting jobs, but with new submission parameters. Our study shows that some runtime execution paths for new features can be disabled by setting proper parameters. When a new bug introduced by a new code change occurs, DRIs may suggest customers to run the job with some specific parameters to avoid the execution of the buggy code segment. For example, a job is failed because the optimizer generates too many vertices to be handled by the job manager. This is a code defect in the optimizer. In order to mitigate the problem, the DRI suggested the customer to resubmit the job with a new job parameter (i.e., the maximal vertex number allowed). In these cases, DRIs are responsible to figure out the causes and the new parameters as a work-around solution.

Hot Fix. For the remaining escalations caused by system code defects, if the code defect can be easily fixed, a hot fix is provided for mitigation. In order to avoid the potential negative regression of the hot fix, DRIs often provide the customers with a private build with the fix. If the fix passes the stress testing and verification, it becomes a formal patch. In our study, only 3.8% (8) of escalations are mitigated with hot fix.

Recover Faulty Operation. About 5.7% (12) of escalations are caused by system operation faults. The mitigation is to recover the faulty operations.

Others. The mitigation solutions for the remaining escalations includes data regeneration by third-party data producer, further re-escalation to AutoPilot team, etc. For example, most issues caused by the design limitations under extreme situations are transient, simply resubmitting the jobs could mitigate them. Their long-term resolution would be provided in the future release cycles.

Finding 4: There are mainly seven categories of mitigation solutions. More than one third of the escalations can be resolved by simply resubmitting jobs. Only 3.8% escalations adopt hot fix.
Implication: It is possible to automate the mitigation operations for most of the escalations. For example, methods for automated recommendation of the mitigation solutions could be developed.

TABLE VI. CLASSIFICATION OF ESCALATION MITIGATION.

Escalation category	Escalation sub-category	Mitigation solution
Customer side fault	Code defect Operation fault Misuse	Instruct customers to refine their code, configuration, submission parameter or data placement
System side fault	Code defect(not regression) Design limitation	Hotfix or nothing to do if it is exposed by extreme conditions Instruct customers to resubmit jobs with new parameters
	Regression	Roll it back
	Operation fault	Recover it
Hardware fault	Machine unhealthy or outage Power off Network device fault Overheating Bit flipping	Instruct customer to resubmit job
	Time drifting	De-commit failure machines or restart a sub-service

C. *Telemetry Data Used for Mitigation*

As described in Section II, ProductA records a lot of telemetry data for troubleshooting. These data includes performance counters of machines, execution traces for each job, and the derived job metrics. Performance counters mainly calculate information about resource usage, throughput, performance, etc. For example, ProductA records for each vertex the time spent on scheduling, queuing, executing, the data size processed by a vertex, the CPU usage of a machine, the memory usage, the network I/O traffic, and so on. Program traces are recorded when the system executes a job, which allow developers to follow the execution trace of a job in ProductA to investigate which job stage failed and why, or to provide detailed information for performance analysis. Different types of telemetry data play different roles when DRIs diagnose escalations. Table VII lists most of the telemetry data ProductA DRIs used.

The diagnosis process roughly consists of the following steps. DRIs often start their diagnosis by identifying a critical path of the problematic job through analyzing the program traces, which could be used to simulate the job execution. The critical path contains bottleneck stages or failed stages. Then, DRIs dive into the bottleneck stage or failed stage, and try to identify some execution outliers or failed vertices. Because there are always hundreds or even thousands of vertices in a single stage, and these vertices are supposed to have similar execution behavior. If some of them behave differently from others, they are suspicious and likely the culprits. If some vertices generate log events (such as exception messages) that do not appear in other vertices, or the performance-related metrics of some vertices deviate largely from the others, these vertices are detected as outliers. For recurring jobs, the telemetry data of the last successful execution are also analyzed as a basis for comparison. Using outlier detection, DRIs can further look into the outlier vertices by checking their log messages or machine-level performance counters (e.g., CPU, memory, disk, and network counters) to identify the potential causes. In most cases, they can find the exceptions in traces or counters. Essentially, such a diagnosis process is a top-down process, guided by a decision-tree to locate the possible causes.

Finding 5: Program traces and performance counters are used in the escalations diagnosis. There is also a clear decision flow in the diagnosis procedure.
Implication: It is possible to design an end-to-end tool to automate the diagnosis by analyzing the telemetry data.

VII. DISCUSSIONS

A. *A Retrospection of Fault Tolerance*

Fault tolerance is considered to be an effective and efficient way to tolerate faults, especially hardware faults. Checkpointing and redundant replication are two broadly used techniques. ProductA relies on multiple data replicas in storage to tolerate data loss, thus it provides high data availability. Clients could access other replicas if the replica it accessed is unavailable. Besides, ProductA also tolerates vertex execution failure. Once a vertex is considered to be failed or timed out, job manager will reschedule it to another machine. If hardware outages happened in the machine executing one upstream vertex, job manager will reschedule the upstream vertex to another machine since the intermediate result is stored at the local disk rather than the persistent storage. Moreover, ProductA also tolerates the "outlier" vertex that caused by unhealthy or overloaded hardware with a duplicated scheduled counterpart. However, there are still 11.4% (24) failures and 8.1% (17) performance slowdown that are due to hardware faults. In this section, we discuss the reasons why the faults cause job slowdown and even failure, in the presence of fault tolerance mechanisms.

Fault-tolerance cannot tolerate all faults. First, fault-tolerant designs cannot tolerate faults in large scale. Fault-tolerance is no free lunch, and there is tradeoff in system design to control the cost. Job manager will kill the job if vertices keep failing and re-executing. System should not continuously tolerate never-successful faults and should not tolerate too many faults. ProductA will kill the job if the number of failed vertices or the number of revocations exceeds a threshold value. There is an example in our study that an important network switch failed, which resulted in more than 250 machines to be unavailable. As both replicas of the accessed data are located at those machines, the job failed.

Second, there is no fault tolerance for job manager, which is reasonable because the failure of single master is unlikely. ProductA aborts the job if job manger fails, which does not provide checkpointing support for job manager; if hardware issue results in job manager failure, the job will fail as a consequence. Similarly, a failure in iob service (JS) will lead to job submission failure, even though it has persistent job status stored in ProductA.

Third, fault-tolerant design cannot tolerate all kind faults. Taking the bit flipping case as an example, it escapes the checksum checking in distributed file system, which ensures

TABLE VII. TELEMETRY DATA USED IN DIAGNOSIS.

Categories	Sub-categories	Examples
Job/vetex specific metrics	Latency metrics	response time, wait time, execution time
	Throughput metrics	degree of parallelism, queue length
	Task IO metrics	read/written bytes, partition number, shuffling size
	Computing resource metrics	token and bonus token usage
Performance counters	CPU usage	average CPU usage percentage
	Memory usage	available memory, paging file
	Disk usage	disk read/write queue length
	Network usage	round trip time, bytes received/sent
	Machine repairing state	Faulty or Healthy
Job/vertex logs	Exception messages	file not found exception
	Log entries of interested execution points	time that the job/stage/vertex get scheduled, queued and executed
	Log entries of interested measurements	data size of a vertex

the replicas to be the same but fails to tolerate the bit flipping error in memory.

Fourth, it is impractical to have fault tolerant design in all components. For example, jobs would fail if the compiler fails because of unavailable input data. Besides, other faults like human operation faults are difficult to be tolerated by system design.

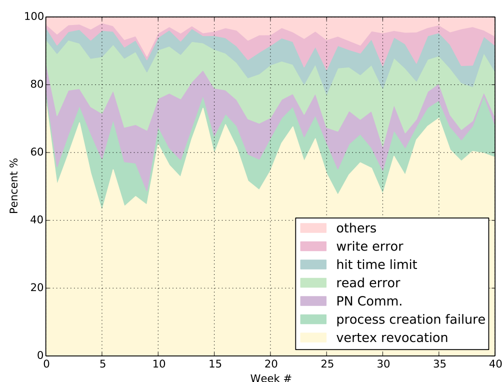

Fig. 6. Top fault tolerant events and their ratio.

Fault-tolerance could waste system resources. We analyze 40 weeks of traces and compute the statistics on the wasted computing time spending on various fault tolerant events. There are about 0.9% to 2.6% of time wasted (i.e., resource wasted) by fault tolerance. Figure 6 further depicts the relative wasted ratio among the top six events: vertex invocation (its upstream vertex is lost), storage read error, storage write error, vertex process creation failure, communication failure with PN, and vertex execution exceeding the time limit. About 42% to 75% of the wasted time are due to vertex invocation, and 9% to 20% due to storage read errors. *Other* reasons of wasted resources include discarded duplicate execution, vertex using too much memory, vertex aborted by job manger, and rarely happened events.

In summary, our study shows that fault tolerance is generally effective and efficient. However, it would sacrifice performance and cannot tolerate all kinds/scales of faults.

B. Implications

Most of the findings and implications we obtain from ProductA are also applicable to other big data computing platforms, such as GFS/MapReduce and HDFS/Hadoop systems. Our findings and implications on live site issues are representative and could be applied to those systems as well. First, such systems are provisioned with the similar commodity hardware in data center, thereby sharing similar hardware faults and similar failure probability. Second, they also have similar system design, and even the same workload characteristics. Although they differ in detailed implementations, they could contain similar system side faults including design limitations. We believe that the experience learned from ProductA would be beneficial to other systems as well. Lastly, all systems provide similar programming interface and languages, therefore customer would produce similar failures. For example, the counterpart of the recursive reducer in Hadoop is *combiner*. The findings and implications on root cause diagnosis and mitigation are also applicable to other systems. With the similar telemetry data such as performance counters and execution traces, those system can apply the same top-down and decision-tree based process in diagnosis and mitigation, because all these systems share the same execution model and storage model, even the operation model. In summary, most of the findings and implications learned from ProductA are also applicable to other Big Data systems as well.

Our study indicates that 70%-90% of TTM time is spent on diagnosis with manual efforts. Therefore, an automated diagnosis tool would relive or save the manual burden. Our findings on the diagnosis of escalations indicate that such an automatic diagnosis tool can be benefited from the telemetry data. Furthermore, our study also helps shape the tool design. Firstly, the study indicates that an abnormal stage either has a large number of failed vertices (tasks) or has a large number of outlier vertices. This suggests that we can apply machine learning techniques to recognize the abnormal stage and also the corresponding problematic features. Secondly, the interested features that DRIs frequently adopted could be derived from the job-related metrics and the associated performance counters. Thirdly, the tool can follow the same two steps as what manual effort does - identifying the abnormal job stage and then reasoning the problematic metrics in that stage. Implementing an automated diagnosis tool that is motivated by this study is an important future work.

C. Threats to Validity

Internal threats to validity Subjectiveness would arise during root cause reasoning because of the large amount of manual effort involved. These threats were mitigated by cross-validation by several team members. If there were different opinions, a discussion was brought up to reach an agreement.

When uncertainty occurred, we contacted the corresponding DRI for confirmation or correction. The classifications could be subjective too; however, we believe they are helpful to derive the findings and implications.

External threats to validity We conducted our study on ProductA only. In Section VII.B, we discuss that some of our major findings could be generalized to other systems. However, it is possible that some detailed results might be specific to ProductA and would not hold for other systems.

VIII. Related Work

Over the years, there have been many empirical studies on the characteristics of failures of software-intensive systems. For example, Seaman et al. [24] collected data from 81 projects across 5 NASA centers. Each center used different defect taxonomies. They developed a unified defect categorization scheme that was compatible with existing data. Li et al.[20] manually collected 709 bugs from two large open source projects (Mozilla and Apache Web Server) and analyzed the bug characteristics. They classified the bugs into different categories (root causes, impacts and software components) and studied the correlation between categories. They found that memory bugs and concurrency bugs accounted for a small portion of bug reports and semantic bugs were the major root causes. Chou et al. [7] presented a study of operating system errors found in Linux and OpenBSD kernels. They found that device drivers had error rates up to three to seven times higher than the rest of the kernel, and that the bugs remained in the Linux kernel an average of 1.8 years before being fixed. Many researchers [10], [2], [29] also analyzed the distribution of failures across modules of a large-scale software system, and observed that a small percentage of the modules is responsible for a large percentage of failures.

Apart from the code defect errors, researchers have found that other types of errors are also major causes of failures. For example, Yin et al. [27] studied 546 real-world misconfigurations for a commercial storage system and four open source systems. They found that a large portion of misconfigurations can cause hard-to-diagnose failures (such as crashes or performance degradation). Gray [14] found that administrator errors were responsible for 42% of system failures in high-end mainframes. Patterson et al. [21] reported that in telephone networks and Internet systems, more than 50% of failures were due to operator errors.

There are also empirical studies on data-parallel programs. Kavulya et al. [18] studied failures in MapReduce programs. There is also a work [17] studied the performance slowdown caused by system side inefficiency. Their studies just take simple workloads rather than production jobs. Xiao et al. [26] conducted study on commutativity, nondeterminism, and correctness of data-parallel programs, and revealed interesting findings that non-commutative reduce functions lead to five bugs. Li et al. [19] studied the failure characteristics in Scope jobs [6], and revealed that exceptional data and mismatched data schema are the major source of job failures, rather than code logic. They advocated a graceful exception handling logic to take care of exceptional data and tooling support to detect the code defects that could be exploited by potential exceptional data. As a complementary, our study studied the system side faults, hardware faults and even human operation faults, and they together provide a comprehensive study on data-parallel programs.

There have been some previous studies in the literature on failures of a data center [22]. For example, Ford et al. studied [11] the data availability of Google distributed storage systems, and characterized the sources of faults contributing to unavailability. Their results indicate that cluster-wide failure events should be paid more attention during the design of system components, such as replication and recovery policies. We studied in a larger scope including not only distributed file system, but also execution engine, and not only the unavailability, but also the job failures and performance issues. Gill et al. [13] presented a large-scale analysis of failures in a data center network. They characterized failure events of network links and devices, estimated their failure impact, and analyzed the effectiveness of network redundancy in masking failures. Vishwanath and Nagappan [25] classified server failures in a data center and found that 8% of all servers had at least one hardware incident in a given year. Both their studies could be helpful to reduce the hardware faults, especially the networking faults. Dinu and Ng [9] analyzed Hadoop behavior under failures of compute nodes, and found that a single failure can result in unpredictable system performance. We share the similar findings that fault-tolerance could slowdown performance and even fail jobs.

Researchers have also studied the statistics on failures recovery or mitigation. For example, in [13], the authors studied the mitigation time distribution and classified them into short and long categories. In our study, more than half of escalations are short-lived issues that can be mitigated by simply resubmitting the job; while the mitigation needs code fix or more time to diagnosis would be long-lived. Zhang et al. [30] performed an empirical study of the bug-fixing time using real industrial projects, and proposed methods to predict the effort required to fix bugs. Benson et al. [5] examined 8,684 reported problems appearing in the forum of a large IaaS provider. They discovered that ten operators were responsible for resolving most problems and that a significant delay of 20-110 hours existed between the initial operator involvement and the problem resolution. They argued that the lessons derived from their study could help design a more efficient support model for cloud computing. Benson et al. [5] examined human activities by operators and support engineers. We do not directly evaluate the individual DRI activity, but we believe that an incentive mechanism in the Big Data ecosystem among engineers, operators and customers, would be helpful to improve daily activities.

IX. Conclusion

This paper has presented one of the first comprehensive studies on quality issue of Big Data computing platform. We have investigated the common symptoms, causes, and mitigation of quality issues of the ProductA system. We have obtained many interesting findings. For example, our study reveals that different types of issues (hardware faults, code defects, human errors, configuration and regression) would occur in Big Data computing. We have also studied the general diagnosis process and the commonly adopted mitigation solutions.

We believe that our findings and implications provide valuable guidelines for future design and maintenance of Big Data platforms. Our study can also serve as motivations for future research on reliable software development as well as efficient maintenance with tooling support.

ACKNOWLEDGMENT

We thank our intern students Yuchao Jin and Meiqing Zhang for their help with the experiments. We would like to thank Sukvinder Singh Gill and Jingren Zhou for their valuable comments. We also thank the product team for their collaboration and comments.

REFERENCES

[1] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G. Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, pages 265–278, 2010.

[2] Carina Andersson and Per Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transaction on Software Engineering*, pages 273–286, 2007.

[3] Apache. Hadoop, 2013. http://hadoop.apache.org/.

[4] Apache. Hadoop distributed file system, 2013. http://hadoop.apache.org/docs/stable/hdfs_design.html.

[5] Theophilus Benson, Sambit Sahu, Aditya Akella, and Anees Shaikh. A first look at problems in the cloud. In *HotCloud*, 2010.

[6] Ronnie Chaiken, Bob Jenkins, Per ke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[8] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] Florin Dinu and T.S. Eugene Ng. Understanding the effects and implications of compute node related failures in hadoop. In *HPDC*, pages 187–198, 2012.

[10] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transaction on Software Engineering*, pages 797–814, 2000.

[11] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, 2010.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[13] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, pages 350–361, 2011.

[14] Jim Gray. Why do computers stop and what can be done about it? 1985.

[15] Michael Isard. Autopilot: automatic data center management. *Operating Systems Review*, pages 60–67, 2007.

[16] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[17] Hui Jin, Kan Qiao, Xian-He Sun, and Ying Li. Performance under failures of mapreduce applications. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 608–609, 2011.

[18] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 94–103, 2010.

[19] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *ICSE*, pages 963–972, 2013.

[20] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Architectural and system support for improving software dependability*, ASID, pages 25–33, New York, NY, USA, 2006. ACM.

[21] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques. Technical report, Berkeley, CA, USA, 2002.

[22] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Barroso. Failure trends in a large disk drive population. In *FAST*, pages 17–29, 2007.

[23] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *FAST*, pages 1–16, 2007.

[24] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. Defect categorization: making use of a decade of widely varying historical data. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 149–157, 2008.

[25] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *SOCC*, pages 193–204, New York, NY, USA, 2010. ACM.

[26] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in mapreduce considered harmful? An empirical study on non-commutative aggregators in mapreduce programs. In *ICSE*, pages 44–53, 2014.

[27] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. SOSP, pages 159–172, 2011.

[28] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.

[29] Hongyu Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, 34(2):301–302, 2008.

[30] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *ICSE*, pages 1042–1051, 2013.