Common Causes and Mitigations of Service Quality Issues in Big Data Computing

Hucheng Zhou[1], Jian-Guang Lou[1], Hongyu Zhang[1], Haibo Lin[2], Haoxiang Lin[1], Tingting Qin[1]

[1]Microsoft Research, [2]Microsoft

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. There is few study in literature on service quality issues in production big data computing platform. In this paper, we present an empirical study on the service quality issues in Dolphin, which is a company-wide multi-tenant big data computing platform in Microsoft, serving thousands of customers from hundreds of teams. Dolphin has a well-defined escalation process (i.e., incident management process), which help customers report and mitigate service quality issues $24/7$.

This paper explores the common causes and mitigations of service quality issues in big data computing. We conduct an empirical study on randomly sampled 200+ live site service quality issues in Dolphin. Our major findings include (1) 21.0% of escalations are caused by hardware faults; (2) 36.2% are caused by system side defects, including 21.0% code defects, 9.5% design limitations and 5.7% operation faults; (3) 37.1% are due to customer side faults, including 11.4% misuses, 11.0% convention violations, 10.0% operation faults, and 4.8% code defects. We also studied the general diagnosis process and the common adopted mitigation solutions; the findings suggest that it is possible to design an end-to-end tool to automate the diagnosis by analyzing the collected telemetry data.

1 INTRODUCTION

Big data computing platforms, including distributed storage systems (GFS [10], HDFS [3]) and distributed data-parallel execution engines (MapReduce [7], Hadoop [2] and Dryad [13]), are prevalently used for storing and processing web-scale data. Dolphin is a distributed big data computing platform that is used within Microsoft for storing and analyzing massive amounts of data; multiple teams such as Ads, Bing, Office 365, Xbox Live, Windows, and Skype use Dolphin for tasks such as web-scale data mining, developing ranking algorithms, and business intelligence. Dolphin has evolved into a company-wide large ecosystem where thousands of customers share hundreds of thousands commodity servers in processing and storage clusters. Tens of thousands of Dolphin "jobs" are executed per day with different workloads and scenarios, while scale is still increasing rapidly.

Dolphin is designed to be fault tolerant where failed jobs are retried as needed. While a variety of hardware and software faults are tolerated gracefully, some jobs chronically fail or suffer performance slowdowns, especially recurring jobs that are submitted hourly, daily, weekly or monthly. Additionally, failed job waste resources: our study of jobs over 90 days showed that job failures accounted for 8% of machine resources. When jobs fail or slow down, the Dolphin team investigates and resolves issues on $24/7$ basis. Guaranteeing service quality for all users with different service level agreements is challenging; instead, Dolphin has a well-defined escalation process that helps customers deal with the live site issues as they come in. Customers escalate issues via email to a support address describing issue experienced by customers, business impact, and relevant troubleshooting details, which are then tracked, prioritized, and processed by the Dolphin engineering team as incidents in an internal tracking system. The engineering team is then responsible for quickly mitigating issue impact, and following up with a post-postmortem and root-cause fix.

This paper studies randomly sampled 210 live site escalations in order to improve the development and operation of big data infrastructure by understanding escalations with their causes and mitigating solutions. Our study aims to address the following two questions:

1. RQ1. What are the common faults that hurt service quality in big data computing?
2. RQ2. What are the common mitigations, and what is helpful in making fast mitigation decisions?

Answers to these questions are generally useful for both customers, system engineers and operators, to improve their daily activities,including system operation, application development, and system design and engineering.

The rest of the paper is organized as follows. Section 2 provides a brief primer on Dolphin. Section 3 describes our study methodology. Section 4 presents the statistics on live site issues. Section 5 studies the common causes to answer research question RQ1, followed by our study results on escalation mitigation in Section 6, which answer the research question RQ2. We discuss the representative of this study in Section 7. Section 8 discusses the related work, and we conclude in Section 9.

2 BACKGROUND

Dolphin includes a distributed file system like GFS and distributed data-parallel execution engine like MapReduce, where the program is written in a language called Parrot [5], which is a hybrid language that consists of declarative SQL-like queries and imperative user-defined functions and is similar to Pig Latin [17] at Yahoo!, Hive [19] at Facebook, and FlumeJava [6] at Google. In Dolphin, a data file stored in Dolphin 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*.

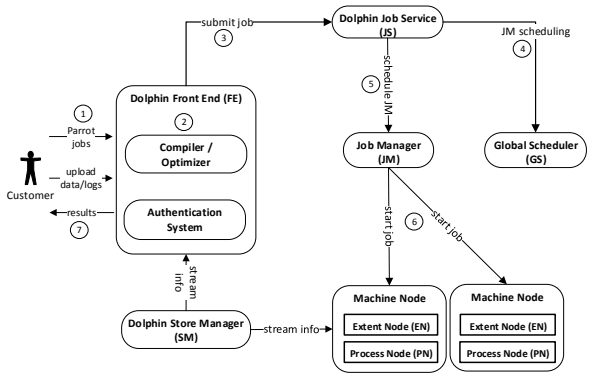

Figure 1: Dolphin work flow.

Figure 1 depicts the brief system framework of Dolphin, which includes front end web services (*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 Dolphin store manager (*SM*) for managing distributed file system metadata similar to the *master* in GFS, and each extent node (*EN*) for storing data extents as local files like that in GFS' *ChunkServer*, a Dolphin 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 manager (*JM*) for coordinating the distributed execution of job vertices, processing nodes (*PN*) for executing job vertices and communicating with the job manager. Each job has a job manger which is actually a special vertex. EN and PN services are deployed to every machine in the data center.

The storage and computing resource in Dolphin are shared and operated by all customer teams in an economic model. Different customer teams are allocated different resource quotas based on their business requirements. The allocated resource quotas for each team are organized as Virtual Cluster (VC)s, which builds an illusion that each team has exclusively access to a dedicated data center. The computing resource unit is called *token*. One customer can only submit his jobs to the VCs his team owned, and the data is shared between VCs by across reference. In this manner, Dolphin can provide a better fair sharing and throttling mechanism, and better VC-wise scheduling where each VC has its own scheduling policy.

Basically, there are two kinds of customer-oriented services, data service for data movement and storage, and job service for job submission and execution which is built on top of data service. Data service is relatively straightforward. Customer requests are dispatched to one FE server and authenticated by AS; if authentication succeeds, FE queries the location of each data extent from SM; and then the customer can directly access the EN through a Dolphin client library. In Dolphin, the work flow of a successful job contains seven steps, which are illustrated in Figure 1: (1) A customer submits a Parrot job with a job priority and the number of tokens required; (2) On receiving the submitted job, FE sends it to the compiler and optimizer, and gets the resulting execution plan; (3) FE sends the compiled job to JS, which queues the job in a corresponding VC queue; (4) The job in the head of queue gets scheduled if the required resources are available, i.e., the number of remaining tokens is larger than the requirement; (5) JS schedules the job manager JM to one machine based on the bookkeeping information maintained in GS; (6) JM creates the physical execution plan (a direct acyclic graph) of the job, where each node is composed of multiple vertices based on a parallelism of the operation. Then JM schedules the ready vertices to different PNs with locality awareness. JM coordinates the vertex execution via heart-beating with PNs. JM would possibly schedule one vertex to another machine to tolerate execution faults or performance slowdown. (7) The customer gets the job completion notice if the job succeeds and the job results are stored as Dolphin streams. Each step would fail or have poor performance. For ease of diagnosis, the system also traces important event logs and stored the daily collected telemetry data as a Dolphin stream for job monitoring and post-mortem analysis, all computation (JM, PN) and storage (SM and EN) components have such event tracing mechanism. To provide a good user experience, Dolphin also provides multiple user interfaces, including a web based UI, a Visual Studio plugin, and so on.

3 METHODOLOGY

3.1 Subjects

We randomly collected 210 escalations that were discussed in 2,196 emails and the corresponding 188 incident tracking records (22 escalations did not have corresponding incident 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 includes the whole life cycle of an escalation, from its submission, the first DRI engagement, and the final mitigation; and incident records are much more structural and friendly for automatic-analysis.

3.2 Design of the Study

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

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

3.3 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 contact 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 Dolphin only. It is possible that some of our findings might be specific to Dolphin and would not hold in other systems. Hence, we do not intend to draw general conclusions for all distributed data-parallel systems. In Section 7, we discuss in details that our findings could be generalized to other systems.

4 What Are The Common Symptom?

We first conduct a study on the common symptoms of service quality issues, by manually collecting the escalation symptoms customers submitted. From the symptom point of view, the escalations are classified into five categories, including connection failure, job slowdown, job failure, wrong result and service unavailable. Table 1 shows the classification statistics that 8.6% (18) of escalations are with connection failures, 27.1% (57) are with job slowdown, 45.2% (95) are with job failures, 8.6% (18) are with wrong result, and the remaining 5.7% (12) are with service unavailable.

Table 1: 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%

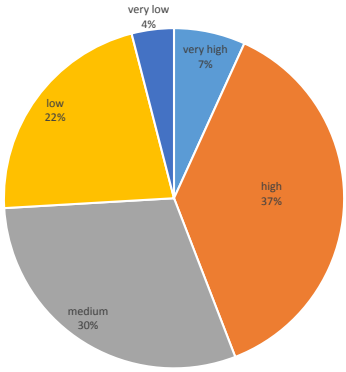

Figure 2: Escalations severity distribution.

We further study the severity of the escalations. When escalating the service quality issue to Dolphin team, customer is obligated to submit the severity level, which determines the DRI engagement priority and the required mitigating time. There are four severity levels from very high, high, medium, low and very low in a descendent order. Escalation with very high level must be engaged less than 15 minutes, and mitigated in 1 hour; while level very low could be engaged with lowest priority and mitigated in one release cycle if it is worth of fixing. The severity level is assessed based on both the business significance of the job or data tier and the impact level of the issue in system side; if the business impact is higher and the impact level is higher, severity level will be higher as a consequence. Figure 2 depicts the severity level 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 and 22% are at low level, and about 4% are very low level.

Table 2: Classification of escalations causes.

Category	Sub-Category	No.	Ratio
Hardware	Subtotal	44	21.0%
fault	System code defect	44	21.0%
System side	Design limitation	20	9.5%
fault	Operation fault	12	5.7%
	Subtotal	76	36.2%
	Code defect	10	4.8%
	Operation fault	21	10.0%
Customer	Misuse	24	11.4%
side fault	Convention violation	23	11.0%
	Subtotal	78	37.1%

Table 3: Classification of hardware faults.

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

5 What Are The Common Causes?

In this section, we classify the escalations from the cause point of view, aiming to answer what are the common causes of escalation.

5.1 Cause Classification

We have reasoned the causes of those 210 escalations manually, and classified them into three categories, including hardware faults, system side faults and customer side faults. Table 2 depicts the detailed categories. For most of the escalations (94.3%), their causes can be successfully classified, with only 5.7% (12) of escalations whose causes are unknown due to the missing evidence. We describe each cause category in details in the following sections.

5.2 Escalations due to Hardware Faults

Modern data center is built with commodity machines that have high probability of failure. Our study shows that hardware fault is one of the most common causes that lead to escalations. We further divide the hardware faults into several sub-categories as shown in Table 3. Some subcategories (such as machine outages, power off, network device fault, and overheating) were also observed and reported by existing studies on hardware faults [18]. It is noteworthy that some machine outages are caused by regular data center maintenance, however, we cannot get the concrete numbers without enough evidence. Our study reveals two unusual hardware faults, namely bit flipping and time drifting.

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 the 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 Parrot script will generate the good data.

Time drifting. Time service, which synchronize time across a network, is critical for the proper operation of Dolphin. There is a case where time stamp of a stream FE reports to be modified at 7:22 pm while the request time was 6:57 pm. The clock of one FE server was out of sync, while soft repairing did not help and that server was decommissioned.

Fault tolerance is considered to be an effective and efficient way to tolerate faults, especially hardware faults. Checkpointing and redundant duplication or replication are two broadly used techniques. Dolphin relies on multiple data replicas in storage to tolerate data loss, thus it provides high data availability; client could access other replica if the replica it accessed is unavailable. Besides, Parrot also tolerate vertex execution failure. Once a vertex is considered to be failed or timed out, job manager will re-reschedule it to another machine. And if hardware outages happened in the machine executing one upstream vertex, job manager will re-schedule the upstream vertex in another machine since the intermediate result is stored at the local disk rather than persistent storage. Moreover, Dolphin also tolerate 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. We continue to study the reason why these hardware faults causes job slowdown and even failure.

Fault-Tolerance could slowdown performance. The existing fault tolerant mechanisms try to continue the job execution when a fault is encountered, however, they could slowdown the system runtime performance since the time spending on failed execution or data access is wasted. The fault tolerance also need machine resource to re-execute the vertex or serve the data access, which could overload the remaining machines. An example is about machine overloaded due to a rack outage with tens of machines lost, thereby all vertices compete the remaining machines, and some vertices (maybe from other jobs) keep timed out and re-executed.

To detect fault, typical data-parallel systems like Dolphin adopt a heart-beating mechanism between job manger and working nodes. An interesting example exposes the ineffectiveness of such heart-beating mechanism. One erroneous machine worked in healthy and unhealthy state rotationally; and job manager assigned one vertex

to it when it is at healthy state, while it then became unhealthy state during the vertex execution; job manager re-schedule the vertex once it detects such fault. However, job manger would schedule vertex to that machine again once it recovers to healthy state. Such fault-tolerant process continues iteratively, and the job performance is largely affected.

Lastly, it is paradoxical that duplicate scheduling may even lead to performance slowdown which is supposed to improve job performance by hiding vertex outlier. It is proactively done by job manger that a vertex is re-scheduled out before the older one fails. Things become subtle if the outlier is caused by data skew, rather than hardware faults. For example, one vertex has data skew and ran much longer than others, duplicate scheduling happens. However, the duplicated vertex still has the same data skew and run slowly. The original vertex would be completed first, thus the resource costed by duplicate scheduling in this case would be wasted. The locality-awareness scheduling makes thing worse, if both original vertex and duplicated vertex were scheduled in the same machine or rack which leads to resource contention and overloaded.

Fault-tolerance cannot tolerate all faults. First, fault-tolerant design cannot tolerate faults in large scale. Fault-tolerance is not 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 continuous tolerate never-successful fault and never tolerate too many faults. Parrot 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; both replicas of accessed data are located at those machines, which failed the job.

Second, there is no fault tolerance for job manager, which is reasonable because the failure of single master is unlikely. Parrot 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, Dolphin job service (JS) failure will lead to job submission failure, even though it has persistent job status stored in Dolphin.

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

Fourth, it is impractical to have fault tolerant design in all components. For example, job failed if compiler failed because of the input data is unavailable, which is due to hardware outage. Besides, other faults like human operation faults cannot be tolerate by system design.

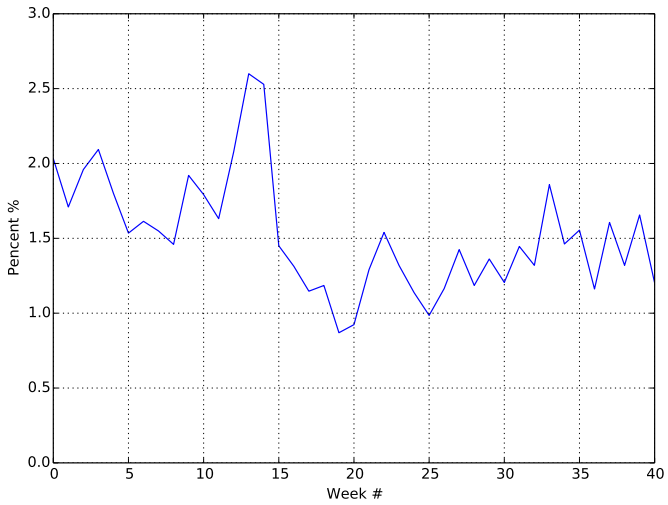

Figure 3: Wasted ratio of computing time on fault tolerance.

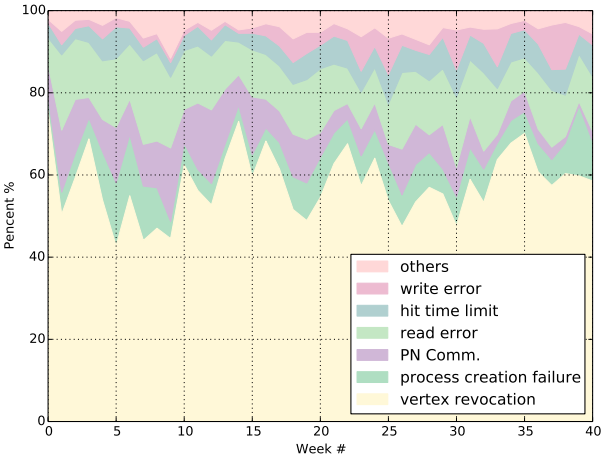

Figure 4: The top fault tolerant events and the corresponding ratio.

We computed the statistics, from 40 weeks traces in all data centers, on the wasted computing time spending on various fault tolerant events, including top six events: vertex invocation due to its upstream vertex is lost, vertex aborted by job manger, storage read or write error, vertex process creation failure, communication failure with PN, and vertex execution time is too long, etc. Figure 3 shows the wasted time ratio spending on those fault tolerant events; there are about 0.9% to 2.6% of time wasted (r.e., resource wasted) by fault tolerance. Figure 4 further depicts the relative wasted ratio among the seven events, where *Others* includes discarded duplicate execution, vertex uses too much memory, and rare happened events.

Table 4: 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%

There are about 42% to 75% of wasted time are due to vertex invocation, 9% to 20% are due to read error, and 2% to 11% are due to *Others*.

Finding 1: 21.0% (44) of escalations are caused by hardware faults.
Implication: Fault tolerance is generally effective and efficient. However, it would sacrifice performance and potentially hurt the service quality in certain cases; and it cannot tolerate all kinds of faults nor all scale of faults.

5.3 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.

5.3.1 System Code Defect

It is unsurprising that code defects in Dolphin account for 21.0% (44) of the escalations, because escalation was send out only if customers consider it to be a system side fault.

Regression. It is surprising that 6.7% (14) of escalations are due to system regression. This is because the roll-out policy adopted in Dolphin. The service-oriented architecture advocates agile and independent development in each sub-service teams; continual changes or new features are rolled out to the online version independently. Currently new features are rolled out via flighting policy like A/B testing, which selects part of the customer jobs or VCs for partial release; and gradually enlarges the scope until fully deployed. To fast mitigate the regression issue, historical versions of system are maintained, 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 is responsible for about three escalations. The defects are mostly trivial code defects. Only two bugs in Parrot optimizer are related to big data computing. One example is that, the optimizer spends more than 25 minutes without any update and the job thereby failed to submit, because the job is large and the optimizer failed to enumerates all possible execution plans to select the one with minimum cost. Another example is that the optimizer tries to repartition the large execution graph, but it finally failed.

Service outage. 5.7% (12) of escalations are caused by service outages, with 83.3% (10/12) of them happened in FE, 8.3% (1/12) in EN and 8.3% (1/12) in compiler. Note that the FE outages were happened in only three time points, with more than one customers escalated such unavailability. Without further more evidence. We assume that such service outages are caused by code defects, because these service are designed to be always available. It is interesting that the corresponding mitigation is just to restart the service.

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

5.3.2 System Design Limitation

In addition to implementation faults, we continue to study the system design limitation. 9.5% (20) of escalations could be avoided with proper system design.

Extreme situation. Six escalations are exposed in extreme situation,where system design limitation is blown up. 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 the machine overloaded and most of vertices failed. Things could become better if the optimizer applies the broadcast join or the job manager throttles the vertices scheduling to a single machine. Another extreme case is related to the extent sealing in SM storage. There is a fixed quota of unsealed extents to be sealed in the background every 10 to 15 minutes because of the expensive operation; however, at one time there are millions unsealed extents, thereby with some extents got unsealed in time, and customer complains the zero-length unsealed extent. Optimizing the time spending on extent sealing in SM would largely alleviate it. A much more interesting example is the job failure 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 turned into two replicas by de-replicating one. If a vertex is accessing one data replica which is under de-replicating, that vertex would repeatedly failed.

By introducing inter-awareness between job scheduling in job manager and the de-replicating process in SM, such case can be avoided.

Resource contention and overloaded. We find that resource contention and overloaded caused eight and six escalations, respectively. These escalations can be avoided by adopting a proper design. One escalation is due to workload imbalance with huge spike in traffic. The contention-aware scheduling could largely alleviate it, as well as the certain performance isolation design. It is not easy for a contention-free or overloaded-free scheduling design, especially considering both data storage (EN) and vertices execution (PN) are executed at the same machine, and there is no knowledge sharing between job manager (JM) and SM. For example, there is an job slowdown happened because some of the vertices are affected by other activities like data downloading request from other customers.

Finding 2: 21.0% (44) of escalations are caused by system code defects with 6.7% (16) system regression, and 9.5% (20) are due to design limitation.
Implication: The fit-for-all design and bug-free implementation of large scale big data infrastructure is infeasible in large scale distributed system. The testing in a broader scope, especially online testing in real production would be helpful to expose the system faults as early as possible.

5.3.3 System Operation Fault

System operation faults include deployment faults, incomplete provision, and library version mismatch. There is one example about a deployment restarts ENs in scale-unit, causing intermediate data loss for long running vertices. An interesting example is that operator updates Parrot runtime which poisons the cached runtime, and then is propagated across many machines. This is a warning sign that we should be cautious and pay more attention to system management operations. Better process management and guidelines are helpful.

5.4 Customer Side Faults

It is surprising that 37.1% (78) of escalations were caused by customer side faults. The statistics are shown in Table 2. The customer side faults include 4.8% (10) code defects, 10.0% (21) operation faults, 11.4% (24) misuse and even 11.0% (23) convention violation system claimed.

5.4.1 Customer Code Defect

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

Buggy and non-optimized code. This type of defects includes optimizable data skew, 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 other else, thus the corresponding vertices become the "outliers" [1]. Code defects also happen to the third-party library, which is invisible and hard to detect.

Table 5: 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	Subtotal	21	10.0%
fault Misuse	Incorrect client configuration	15	7.1%
	Improper submission parameter	9	4.3%
	Subtotal	24	11.4%
Convention violation	Out of system endurance	17	8.1%
	Beyond system capability	3	1.4%
	Delay time window	3	1.4%
	Subtotal	23	11.0%

The advanced language feature provided by Parrot could also introduce defects. Parrot allows customer to write user-defined recursive reducer, which provides partial aggregation optimization [21], like the *combiner* in Hadoop. 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 that the recursive reducer must be associative and commutative, since it would be executed more than once. However, in this example customer did not know such semantics constraints, and obtained the wrong result.

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

Finding 3: 4.8% (10) of escalations are caused by customer code defect, which includes buggy and non-optimized code, as well as inhibitive programming style.
Implication: Such code defects are hard to diagnose. It is better to detect the error-prone patterns, optimize the data skew, and enforce the programming style much earlier during compilation.

5.4.2 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.

5.4.3 Misuse

We treat incorrect client configuration and improper submission parameter as misuse; they are mitigated by resubmission with new configuration or parameters.

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

Improper submission parameter. 4.8% (10) of escalations are due to improper job submission parameter. 40.0% (4) of them could be mitigated by increasing the resource quotas to meet the job SLA. Job performance varies (maybe largely) depending on whether or not the data center is busy, which affects the ability to get the "free" resources. Sometimes the job latency could satisfy the SLA even with the same smaller quotas, while performance slowdown happens if the data center is very busy. It would be better if tools can decide the right number of resource quotas as small as possible to satisfy job SLA. Another interesting job submission parameter is about the largest number of scheduled vertices across the Vlan to control the cross Vlan traffic by job manager. This will delay the vertex scheduling latency especially in job critical path. Besides, system also provides a option to control the unavailable ratio of input data, since it is not uncommon that there is negligible data loss in big data computing, which can tolerate it with almost the same result.

Finding 4: 10.0% (21) of escalations are are caused by customer operation faults, and 11.4% (24) are due to customer misuse, with 21.4% (45) in total.
Implication: Better training and process management are helpful to avoid misuse; and automatic configuration or adjusting is a promising direction to reduce the configuration faults.

5.4.4 System Convention Violation

System is always designed with certain assumptions and conventions. Such design assumptions should be clearly communicated to and agreed by customers. Our study finds that 11.0% (23) of escalations are due to violation of such conventions.

Out of system endurance. 8.1% (17) of escalations occurred due to out of system endurance. 7 of them are out of endurance in job manager, such as any vertex execution consumes more than 6 GB memory, or the execution exceeds a threshold time. The job manager will kill these jobs to control the contention in shared resource or punish the non-optimized or incorrect code. Most of these cases are due to data skew or computation skew, which could be optimized by customer. For example, customer can change the keys to reduce or join for more balanced partition, or apply selection and projection in query optimization to reduce the data volume as early as possible. FE also has the throttling mechanism to control the request rate from customers. It is noteworthy that most of them are due to automatically generated requests by client side programs.

Beyond system capability. There are 1.4% (3) of escalations that are beyond the system capability. For example, compiler has the internal limitation on the number of partitions, SM has limited the number of simultaneous stream creations, and Parrot runtime has limitation on the data volume to be sampled. There is one extreme case that customer created more than 1 million streams at a time, which is disallowed by system.

Delay time window. There is a 15 minutes delay time window between the creation of data stream and the first use. If customer accessed the data in such time window, an error complaining the instable data stream would occur. It is actually because that the job that generated the input data is delayed and behind the time schedule.

Finding 5: 11.0% (23) of escalations are are caused by system assumption violation.
Implication: It is better to detect such violation patterns as early as possible. Meanwhile, better failure messages could help customer understand and diagnosis the problems.

6 From Escalation to Mitigation

When a service quality issue is escalated to Dolphin team, DRIs are responsible to provide a mitigation solution in time. In this section, we study the mitigation they applied and investigate what kind of information they used for diagnosis.

6.1 Mitigation Categorization

Compared to traditional bug fixes, the aim of mitigation is to recover the system and resume the impacted jobs as quickly as possible to meet the SLAs. A mitigation may not need to be a thorough root cause fix. On the contrary, we only need a work-around solution for most cases. In our study, we found that there are mainly seven categories of mitigation actions that DRIs have adopted. Table 6 depicts the detailed categories of mitigation solutions, with only 2.38% (5) of escalations whose mitigation is

unknown due to lack of explicit description in emails and internal bug records.

Refine customer code, configuration or submission parameter. As mentioned in Section 5, about 37.1% escalations are caused by the faults at customer side. From DRIs' perspective, there is no any fault in Dolphin and no actual mitigation adopted in system side. In current practice, DRIs instead provide the diagnosed causes to customers, and suggest customers to refine their code, configuration, submission parameter or data placement, and so on.

Resubmit jobs. About 21.0% escalations are caused by hardware faults, and most of them (19.1%) are intermittent faults; simply resubmitting the job for job failure could automatically mitigated it, and there is nothing to do for job slowdown. In fact, all machines in our data centers are managed by AutoPilot [12] which detects and recovers from 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 5, 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 program: 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. Here is a real escalation. Customer reports that their job ran very slowly, which was caused by a disk fault which reading 2G bytes costed more than 30 minutes. Such "weird" machines are likely to become faulty soon [18], DRIs need to label the machine as a bad state and de-commit the machine. Meanwhile, customer 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 Parrot runtime versions are stored in the system, and there is no explicit rolling back option; instead, customer resubmits the job with a parameter to configure it with the previous runtime version. 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 exsection paths of the run-time for new features can be disabled by setting proper parameters. When a new bug introduced by a new code segment 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 of optimizer. In order to mitigate the problem, the DRI suggested the customer to resubmit the job with a new job parameter (r.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 fault. The mitigation is to simply recover the faulty operation.

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 design limitation under extreme situations are transient, simply job resubmission could mitigate them. Long-term resolution for some system code defects and design limitations would also be provided in the future release cycle.

Finding 6: There are mainly seven categories of mitigation solutions. More than one third escalations can be resolved by simply resubmitting jobs. Only 3.8% escalations adopt hot fix.
Implication: It is possible to automate the mitigation operations of most escalations.

6.2 Telemetry Data Used for Mitigation

The above mitigating solutions are relative straightforward and the TTM time is mainly spend on causes diagnosis. The fast diagnosis would be helpful to reduce the TTM time. As described in Section 2, Dolphin records a lot of telemetry data for troubleshooting including performance counters of machines, execution traces for each job. Performance counters are a set of special-purpose counters built into modern microprocessors, operating systems, and applications to keep the counts of system activities within computer systems. Performance counters mainly count the information about resource usage, throughput, performance, etc. For example, Dolphin records the data size processed by a vertex, the processing time of a vertex, the CPU usage of a machine, the memory usage of a machine, and so on. Program traces are recorded

Table 6: Classification of escalation mitigation.

Escalation category	Escalation sub-category	Mitigation solution
Customer side fault	Code defect Operation fault Misuse Convention violation	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, nothing to do for job slowdown
	Time drifting	De-commit failure machines or restart a sub-service

when the system executes a job, which allow developers to follow the execution of a job in the system to investigate why a code path has failed, or to provide detailed information for performance analysis. Different types of telemetry data play different roles when DRIs diagnose escalations. Table 7 lists most of telemetry data they used. Note that some application level performance counters are calculated from the log messages.

The diagnosis process roughly consists of the following steps. DRIs often start their diagnosis from 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 outlier vertices. Because there are always 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 that do not appear in other vertices, or the performance-counter values of some vertices are varies largely with others, these vertices are detected as outliers. For recurring jobs, the telemetry data of the last successful execution are also analyzed as basses for comparison. With outlier detection, DRIs will 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 that exception messages in traces or counters. Essentially, such diagnosis process is top-down guided by a decision-tree to exclude the impossible causes.

For some escalations, local simulation were conducted to diagnose their root causes, r.e., downloading all vertex execution related info into local disk and re-run it as a simulation.

Finding 7: Program traces and performance counters are used in the escalations diagnosis. And a clear decision flow exists in the diagnosis procedure.
Implication: It is possible to design an end-to-end tool to automate the diagnosis by collecting and analyzing the program traces and performance counters.

7 DISCUSSION

Most of the findings and implications we obtained from Dolphin are suitable for 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 Dolphin would be beneficial to other systems as well. Lastly, all systems provide similar programming interface and languages, therefore customer would produce the 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 Dolphin are also applicable to other systems as well.

Table 7: Telemetry data used in diagnosis.

Categories	Sub-categories	Examples
Application specific performance counters	Request latency related counters	response time, wait time, execution time
	Throughput related counters	# of request per second, queue length
	Vertex IO related counters	read/written bytes, partition number, shuffling size
	Computing resource related counters	token usage
Operating system 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
Program logs	Exception messages	file not found exception
	Log entries of important execution points	entering and leaving a stage
	Log entries of important measurements	data size of a vertex

8 Related Work

There have been some previous studies in the literature on the network or hardware failures [18] of a data center. For example, Ford et al. studied [9] 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. [11] 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 [20] classified server failures in a data center and found that 8% of all servers had at least one hardware incident in a given year. They also found that the distribution of successive failure on a server fits an inverse curve. Both their studies could be helpful to reduce the hardware faults, especially the networking faults. Dinu and Ng [8] analyzed Hadoop behavior under failures of compute nodes, and found that a single failure can result in unpredictable system performance. They believed that this problem was caused by unrealistic assumptions about task progress rates made by Hadoop. We share the similar findings that fault-tolerance could slowdown performance and even fail jobs.

There are also empirical studies on data-parallel programs. Kavulya et al. [15] studied failures in MapReduce programs. There is also a work [14] studied the performance slowdown caused by system side inefficiency. Their studies just take simple workloads rather than production jobs. Li et al. [16] studied the failure characteristics in SCOPE jobs, 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 also studied the system side faults, hardware faults and even human operation faults, and they together provide a comprehensive study on data-parallel programs.

Researchers have also studied the failures recovery or mitigation. For example, in [11], the authors computed the repair time as the time between a device's failure notification and the time it is reported as being back online. They found that load balancers experienced short-lived failures and inter-data center links took the longest time to repair. In our study, more than half of escalations can be mitigated by simply resubmitting the job, which is short-lived issues; while the mitigation needs code fix or more time to diagnosis would be relative long-lived. Zhang et al. [22] 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. [4] 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. We do not directly evaluate the individual DRI activity, but believe that an incentive mechanism in the big data ecosystem among engineers, operators and customers, other than specific techniques, would be helpful to improve almost all kinds of daily activities, including not only system operation, but also application development, system design and engineering, etc.

9 CONCLUSION

This paper has presented the first comprehensive study on escalations to mitigate service quality issue in production big data computing platform. We aim to answer research questions such as "what are the common causes of the issues in the system side"? and "how to diagnose and mitigate the issues in practice?". We studied more than 200+ escalation records. The results reveal that different types of issues (hardware faults, code defects, human errors, configuration and regression) occurred in big data computing. Although fault-tolerance works well on tolerating most of the hardware faults and hiding the vertex latency for most cases, it would slowdown the job execution and even fail the job in certain extreme situations. The study also tries to bridge the gap between escalation symptom and its (root) causes, and the gap between (root) causes and the mitigation or resolution. We believe that our findings and implications provide valuable guidelines for future design and maintenance of big data infrastructure, and also serve as motivations for future researches on reliable program development and efficient escalation processing with tooling support.

REFERENCES

[1] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A. G., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in map-reduce clusters using mantri. In *OSDI* (2010), pp. 265–278.

[2] APACHE. Hadoop, 2013. http://hadoop.apache.org/.

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

[4] BENSON, T., SAHU, S., AKELLA, A., AND SHAIKH, A. A first look at problems in the cloud. In *HotCloud* (2010).

[5] CHAIKEN, R., JENKINS, B., KE LARSON, P., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB 1*, 2 (2008), 1265–1276.

[6] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: easy, efficient data-parallel pipelines. In *PLDI* (2010), pp. 363–375.

[7] DEAN, J., AND GHEMAWAT, S. MapReduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[8] DINU, F., AND NG, T. E. Understanding the effects and implications of compute node related failures in hadoop.

[9] FORD, D., LABELLE, F., POPOVICI, F., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *OSDI* (2010).

[10] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP* (2003), pp. 29–43.

[11] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM* (2011), pp. 350–361.

[12] ISARD, M. Autopilot: automatic data center management. *Operating Systems Review* (2007), 60–67.

[13] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), pp. 59–72.

[14] JIN, H., QIAO, K., SUN, X.-H., AND LI, Y. Performance under failures of mapreduce applications. In *CCGRID* (2011), pp. 608–609.

[15] KAVULYA, S., TAN, J., GANDHI, R., AND NARASIMHAN, P. An analysis of traces from a production mapreduce cluster. In *CCGRID* (2010), pp. 94–103.

[16] LI, S., ZHOU, H., LIN, H., XIAO, T., LIN, H., LIN, W., AND XIE, T. A characteristic study on failures of production distributed data-parallel programs. In *ICSE* (2013), pp. 963–972.

[17] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD* (2008), pp. 1099–1110.

[18] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure trends in a large disk drive population. In *FAST* (2007), pp. 17–29.

[19] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTONY, S., LI, H., AND MURTHY, R. Hive – a petabyte scale data warehouse using Hadoop. In *ICDE* (2010), pp. 996–1005.

[20] VISHWANATH, K. V., AND NAGAPPAN, N. Characterizing cloud computing hardware reliability. In *SOCC* (New York, NY, USA, 2010), ACM, pp. 193–204.

[21] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP* (2009), pp. 247–260.

[22] ZHANG, H., GONG, L., AND VERSTEEG, S. Predicting bug-fixing time: an empirical study of commercial software projects. In *ICSE* (2013), pp. 1042–1051.