Overdraft: Graceful Handling of ENOSPC in Multi-Tenant Storage Infrastructure

Ahmed Dajani
adajani@iastate.edu
Iowa State University
Ames, IA, USA

Wei Xu
weixu@iastate.edu
Iowa State University
Ames, IA, USA

Mai Zheng
mai@iastate.edu
Iowa State University
Ames, IA, USA

Andrea Arpaci-Dusseau
dusseau@cs.wisc.edu
University of Wisconsin–Madison
Madison, WI, USA

Remzi Arpaci-Dusseau
remzi@cs.wisc.edu
University of Wisconsin–Madison
Madison, WI, USA

Abstract

Handling out-of-space errors remains a persistent challenge for data-intensive applications, often resulting in application crashes and misleading error messages. We present a study showing how applications react poorly to space exhaustion across diverse storage systems, revealing long error chains and inconsistent behavior. Motivated by the study and the dynamic, multi-tenant nature of modern storage infrastructures, we introduce Overdraft, a system that detects, manages, and mitigates storage capacity violations through dynamic space reallocation. Our eBPF-based prototype demonstrates low overhead and seamless integration with local and distributed file systems, enabling efficient thin provisioning while avoiding the reliability pitfalls of out-of-space conditions.

1 Introduction

Writing correct data-intensive programs remains a challenge in the modern datacenter. Such programs, including databases [6, 9, 29, 32, 47], distributed storage systems [7, 22, 36], key-value stores [1, 11, 14, 35], and search engines [17, 26, 43], not only must support a wide range of functionality, but also need to handle any and all errors that might occur during execution. As many previous studies have shown [12, 19, 28, 33, 34, 40, 49], correctly handling errors during runtime is hard – and even the most well-tested of applications do not handle the wide range of possible errors correctly [19, 21].

One particularly vexing error family to handle arises when an application runs out of storage space. The classic error in this domain is ENOSPC ("There is no free space remaining on the file system containing the file") but also includes other related errors that can arise (e.g., EDQUOT – "The user's quota of disk blocks on the file system containing the file is exhausted"). These errors are challenging to handle because they can arise at any time – any operation that needs additional disk space to complete can trigger them – and that worse, as there is no space left, correct remediation is challenging or impossible.

In this paper, we first perform a study to demonstrate the complications of handling this family of errors. Specifically, we study how two applications (a database – CouchDB, and a object-store client – SwiftClient) running on an array of storage systems (Ext4, ZFS, CephFS, Swift) react when the storage system runs out of space. By controlling storage capacity carefully and triggering limit violations systematically, we can determine which errors manifest under space constraint, and further see how these applications react in the presence of such errors.

Our findings are as follows. First, when running out of space, we find that applications often fail, and worse, report error messages that are confusing or misleading. For example, CouchDB reports errors such as "couchdb.NoNode ReacheableException", "internal_server_error" or "Database does not exist", none of which are appropriate given the error injected. Moreover, we find that a root cause of this confusion is the incredibly long error chain that is common in these types of applications. Finally, there is a general lack of consistency in which specific errors are triggered and thus an increase in complexity in error handling is required.

We next observe that in the datacenter, running out of space is a problem that can, in many cases, be avoided altogether, due to the abundance of availability of storage. As opposed to the classic single-machine single-user environment, data-intensive applications often run in shared, multi-tenant environments (e.g., the cloud or other virtualized infrastructures); thus, either before or when a disk-space limitation is reached, additional storage can be requested and made available, avoiding the hard-to-handle ENOSPC and related problems described above.

Based on the observation, we introduce *Overdraft*, a system to track space usage underneath such applications and dynamically adjust space needs (within prespecified limits) to avoid confusing out-of-space issues in the general case. Overdraft includes a *detector* to track the storage usage and detect

The 3rd Workshop on Hot Topics in System Infrastructure (HotInfra'26), co-located with ISCA'26, June 28, 2026, Raleigh, NC, USA

capacity violations in three modes each with different tradeoffs. Once a violation is detected, a *policy maker* specifies the handling policies which may either use unallocated space or borrow unused quota from other tenants to improve overall utilization. Finally, based on the policies, a *capacity extender* uses system-specific plugins to extend storage capacity and thus handle violations gracefully. Our preliminary prototype is promising: the eBPF-based ENOSPC detector incurs little overhead, and the plugins for local file systems and CephFS can extend the persistent volume without disrupting the application container. Overdraft can thus enable applications be configured to run with minimal storage requirements – avoiding the potential costs of overprovisioning – while also avoiding the usual penalties of such thin provisioning, i.e., confusing errors, crashes, and downtime.

2 Understanding ENOSPC Issues

Study Methodology. As shown in Table 2 and Table 3, our study involves four representative storage systems with different multi-tenancy features and two popular applications. We build a 3-stage workflow to apply concurrent workload operations, control write sizes carefully to trigger violations, and collect logs for postmortem diagnosis.

Observations. Table 1 shows a few examples of errors across system layers. We summarize general observations below:

First, *capacity limit violations can crash applications with various types of error messages, many of which are not helpful.* For example, when running YCSB on CouchDB with a limited persistent volume, the workloads may crash with an exception "`couchdb.NoNode ReacheableException`". Similarly, accessing CouchDB via HTTP may trigger "`internal_server_error`" or "`Database does not exist`" errors, which might be confusing to the end users.

Second, *there is a deep error propagation chain across levels, and different low-level errors may lead to the same user messages.* For example, an `ENOSPC` error returned by a `write` system call may lead to an "`no match of right hand value {error,enospc}`" error in the BEAM [37] VM of CouchDB, which may eventually lead to a (confusing) error message at the user interface (e.g., "`internal_server_ error`").

Third, *low-level errors are informative, but they are not always consistent across systems.* As shown in Table 4, we observed ten POSIX errors in total. Both Ext4 (E) and ZFS (Z) may generate an `ENOSPC` error when the capacity limit is violated, but CephFS (C) with quota support may generate an `EDQUOT` error instead. Moreover, neither `ENOSPC` nor `EDQUOT` may be triggered when Swift's quota limit is violated.

In addition, we observed one interesting issue in Swift's quota mechanism: *Regular users may exceed their quota limits (temporarily).* This is likely due to the eventual consistency model used in Swift object store, which may lead to stale storage status on the OpenStack controller. Overall, our study suggests the complications of managing storage infrastructures and calls for fine-grained error detection and capacity control in multi-tenant environment.

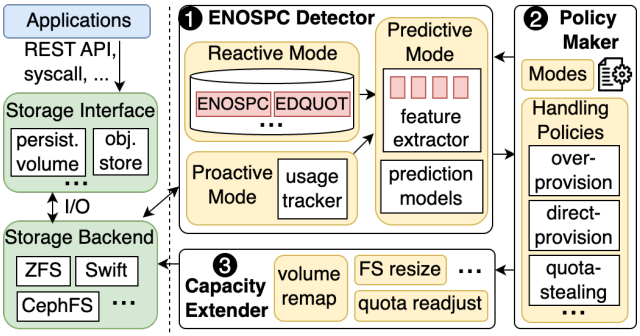

Figure 1. Overview of Overdraft

3 How to Handle ENOSPC

3.1 Design Overview

Based on the observations (§2), we introduce `Overdraft`, a framework to address capacity violations gracefully in multi-tenant storage infrastructure. As shown in Figure 1, the framework consists of three main components as follows:

ENOSPC Detector. To address the capacity violations, it is necessary to identify the potential issue first. To this end, ENOSPC Detector includes three detection modes with different tradeoffs. First, the *Reactive Mode* monitors the return values of write-related system calls in the target system via eBPF [16] and detect relevant low-level errors reactively when they occur. Once detected, it sends a signal to the next component (*Policy Maker*) for handling. Note that different sets of low-level errors need to be captured for different storage systems as revealed by our study (Table 4). Also, other metadata (e.g., timestamp, process ID, syscall parameters) can be recorded for building prediction models.

Second, the *Proactive Mode* tracks the capacity usage of the target system and raises a signal proactively when a predefined threshold (e.g., 90% of the allowed limit) is reached. This mode gives up full space utilization for simplicity.

Third, the *Predictive Mode* leverages the monitoring capability of the other two modes to continuously collect usage patterns, extract short-term and long-term error features, and build/update models for predicting capacity violations in advance. Like MVTRF [46], we focus on supervised learning (e.g., decision trees [27]) because our 3-stage workflow (§2) can reliably generate capacity violation errors for the initial training and modeling. But we expect other AI techniques including LLMs are potentially applicable too.

Policy Maker. This component specifies the detection modes and relevant parameters (e.g., threshold for *Proactive Mode*). Moreover, it defines a set of policies for handling detected capacity violations. For example, simply over-provisioning storage capacity for each individual tenant is one straightforward policy to tolerate capacity violations to some extent, with the downsize of potentially wasting over-provisioned space. Alternatively, in case there are unallocated capacity

Levels	CouchDB (w/ persistent volume)	SwiftClient (w/ object store)
User Message	*"couchdb.NoNodeReacheableException", "internal_server_error", "Database does not exist"*	*"Error: Unable to copy the object", "Object COPY failed 413 Request Entity Too Large 'Upload exceeds quota.'"*
App. Log	*"no match of right hand value {error,enospc}", "no match of right hand value {error,edquot}"*	*"proxy-server ⟨clientaddr/time⟩ PUT ⟨objectdir⟩ HTTP/1.0 413" "proxy-server ⟨clientaddr/time⟩ COPY ⟨objectdir⟩ HTTP/1.0 413"*
Syscall	ENOSPC, EDQUOT, EAGAIN, EROFS, ENOENT	EAGAIN, ENOTDIR, ESPIPE, ENOTTY, EINVAL

Table 1. Examples of Errors Observed at Different Levels.

System	Description	Usage
Ext4 [25]	default file system on Linux	persistent volume of container
ZFS [3]	file system w/ dataset delegation	
CephFS [39]	distributed file system w/ quota	
Swift [30]	object storage system w/ quota	object store

Table 2. Target Storage Systems & Usage.

Application	Wkld Ops	Description
CouchDB [11]	YCSB-A [10]	an update heavy workload
	YCSB-D [10]	delete, insert, and read records
	curl -put	add a document via HTTP
SwiftClient [31]	PUT	create/update object
	POST	create/update object metadata
	COPY	copy objects
	manual	upload objects manually via GUI

Table 3. Applications & Workload Operations.

Error	Meaning	E	Z	C	S
ENOSPC	No space left on device	X	X		
EDQUOT	Disk quota exceeded			X	
EAGAIN	Resource temp. unavailable	X	X	X	X
ENOENT	No such file or dir.	X	X	X	X
ENOTTY	Inappropriate I/O control op.				X
ESPIPE	Invalid seek				X
ENOTDIR	Not a directory				X
EINVAL	Invalid argument				X
EROFS	Read-only filesystem	X	X	X	
EACCES	Permission denied	X	X	X	

Table 4. POSIX Errors Observed on Different Backends (E: Ext4, Z: ZFS, C: CephFS, S: Swift).

available in the system (which is common in large-scale infrastructure), provisioning additional resource to the application on-demand to tolerate the violation based on the signal raised by ENOSPC detector (i.e., *direct-provisioning*) is more space-efficient. Moreover, similar to the classic work-stealing in multithreaded programming [2], we can steal or borrow unused quota from other tenants to make the overall storage utilization more efficient (i.e., *quota-stealing*).

Quota-borrowing semantics function as follows: Borrowed quota is revocable. If a lending tenant's usage approaches its allocated limit, the Policy Maker will reclaim the borrowed capacity and notify the borrowing tenant to either relinquish data or reduce their usage back to the original quota. During periods of global storage pressure—when there is neither unallocated capacity nor a lendable surplus available—the Policy Maker will proportionally limit new writes from all tenants who exceed their base quotas. This approach prevents any single tenant from monopolizing shared resources.

To safeguard against tenants exhibiting non-compliant behavior, a strict upper-bound cap is enforced for each tenant. A tenant cannot borrow beyond this cap, regardless of the available surplus. Additionally, instances of tenants triggering this cap are logged for operator review.

Note that additional policies may be added and all policies are limited by an upper-bound to ensure fairness and security in the multi-tenant environment. Overdraft differs from existing mechanisms in its scope and unification. For example, Kubernetes LimitRange and ResourceQuota enforce static per-namespace caps but provide no dynamic reallocation when limits are hit. LVM thin provisioning extends block-layer capacity but is unaware of application-level quota semantics or tenant borrowing. CephFS quota commands adjust directory quotas manually but require operator intervention and have no cross-tenant policy layer. Overdraft unifies cross-layer detection, tenant-aware policy enforcement, and system-specific extension into a single framework, enabling automated, graceful handling that none of these mechanisms provides individually.

Capacity Extender. This component includes a set of plugins for different storage systems to extend the capacity for individual tenants based on the policies specified by *Policy Maker*. Note that the plugins are system-specific as they depend on the target system's multi-tenancy feature. For example, the Ext4 plugin extends the file system via `resize2fs` [38]. Similarly, the CephFS plugin extends a directory with quota limit by adjusting its extended attribute.

3.2 Preliminary Prototype & Evaluation

We have built a preliminary prototype which supports eBPF-based ENOSPC detection (i.e., *Reactive Mode*) and includes capacity extension plugins for local file systems (Ext4, ZFS) and CephFS. The eBPF-based detector introduces tracepoints to write system calls and checks the return values to capture representative low-level errors as revealed in our study (§2). The local file system plugins leverage LVM to manage and extend logical volume groups to provide additional block space before resizing the file systems on it. The CephFS plugin resets the extended attributes (`ceph.quota.max_files` and `ceph.quota.max_bytes`) of the corresponding directory to extend its quota accordingly.

We conducted preliminary experiments on a computer equipped with an Intel(R) Xeon(R) E5530 processor (4 cores, 8 threads), 12GB of RAM, and 465.8GB of local HDD storage. Similar to §2, three storage systems (i.e., Ext4, ZFS, CephFS) were mounted as persistent volumes for supporting CouchDB under YCSB workloads (A and D). We measured the operations per second (ops/s) and calculated the average of ten runs.

The eBPF-based detector (Reactive Mode) incurs little overhead, ranging from 0.31% to 3.08% across the evaluated storage backends. The Capacity Extender plugins operate with negligible impact on running workloads: the Ext4 and CephFS extensions modify only metadata (block group descriptors and quota extended attributes, respectively) without moving existing data, and both complete transparently without restarting the container. The Policy Maker decision path is similarly lightweight, as policy selection reduces to a threshold comparison against monitored usage. These preliminary results demonstrate the potential of graceful handling of ENOSPC errors without causing confusing messages or severe downtime.

4 Discussions & Future Work

Prediction for ENOSPC Issues. The Predictive Mode of ENOSPC Detector (§3.1) is largely inspired by the rich literature on studying storage failure patterns [15, 41, 48] and applying AI techniques to address system issues [5], including HDD/SSD failure prediction [4, 18, 20, 23, 24, 46], anomaly detection [13, 42, 44, 45], etc. One challenge we face is the lack of large-scale ENOSPC datasets. To mitigate the open challenge, we will leverage the automated workflow (§2) to generate more comprehensive error traces to facilitate follow up research including fine-tuning LLMs for error handling.

Cost Modeling & Policies. The Policy Maker component (§3.1) only considers resource allocations from the technical perspective. In practice, similar to the overdraft protection services provided by banks, there are always limits and fees need to be considered for the extra resources, which may involve sophisticated cost modeling that can affect the design of protection policies. While these non-technical factors are outside the scope of our work, we believe the Overdraft framework can serve as a foundation for exploring more complicated policies in practice, similar to Leopard [8].

Large-Scale Multi-Tenancy. As of this writing, we have only studied the ENOSPC issues and conducted experiments at a well-controlled small scale with limited multi-tenancy. On production infrastructures, the number of concurrent tenants may be multiple orders of magnitude larger which will likely incur much higher variety in terms of storage requests and thus lead to more challenging ENOSPC scenarios. We will investigate this further in collaboration with the broader infrastructure community.

Acknowledgments

The authors thank the anonymous reviewers for their invaluable feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS#1943204, #2402858, and #2402859. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] Apache. [n. d.]. Apache Cassandra | Apache Cassandra Documentation. https://cassandra.apache.org/_/index.html. Accessed: June 5, 2025.

[2] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[3] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, Vol. 215. 1.

[4] Mirela Madalina Botezatu, Ioana Giurgiu, Jasmina Bogojeska, and Dorothea Wiesmann. 2016. Predicting disk replacement towards reliable data centers. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 39–48.

[5] Jalil Boukhobza, Suren Byna, Yuan-Hao Chang, Shadi Ibrahim, Safdar Jamil, Youngjae Kim, Chieh-Lin Tsai, Mai Zheng, and Amelie Chi Zhou. 2026. A Survey on the Use of AI/ML for Storage System Design and Optimization. (2026).

[6] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. 2019. *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.".

[7] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. 2018. PFault: A general framework for analyzing the reliability of high-performance parallel file systems. In *Proceedings of the 2018 International Conference on Supercomputing*. 1–11.

[8] Tingjia Cao, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Tyler Caraza-Harter. 2025. Making Serverless {Pay-For-Use} a Reality with Leopard. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 189–204.

[9] Prabhakar Chaganti and Rich Helms. 2010. *Amazon SimpleDB developer guide.* Packt Publishing Ltd.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[11] CouchDB. [n. d.]. CouchDB. https://couchdb.apache.org/. Accessed: June 5, 2025.

[12] Xiaoning Ding, Hai Huang, Yaoping Ruan, Anees Shaikh, and Xiaodong Zhang. 2008. Automatic Software Fault Diagnosis by Exploiting Application Signatures.. In *LISA*, Vol. 8. 23–39.

[13] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[14] etcd authors. [n. d.]. etcd. https://etcd.io/. Accessed: June 5, 2025.

[15] Om Rameshwar Gatla, Duo Zhang, Wei Xu, and Mai Zheng. 2023. Understanding persistent-memory-related issues in the linux kernel. *ACM Transactions on Storage* 19, 4 (2023), 1–28.

[16] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. 2024. The eBPF Runtime in the Linux Kernel. *arXiv preprint arXiv:2410.00026* (2024).

[17] Google. [n. d.]. Google. https://www.google.com/. Accessed: June 5, 2025.

[18] Yunfei Gu, Chentao Wu, and Xubin He. 2024. Exploit both {SMART} Attributes and {NAND} Flash Wear Characteristics to Effectively Forecast {SSD-based} Storage Failures in Clusters. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1101–1117.

[19] Haryadi S Gunawi, Cindy Rubio-González, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct.. In *FAST*, Vol. 8. 1–16.

[20] Jin Yong Ha, Sangjin Lee, Heon Young Yeom, and Yongseok Son. 2024. {RL-Watchdog}: A Fast and Predictable {SSD} Liveness Watchdog on Storage Systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1083–1100.

[21] Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, and Jonathan Cook. 2022. A study of failure recovery and logging of high-performance parallel file systems. *ACM Transactions on Storage (TOS)* 18, 2 (2022), 1–44.

[22] Runzhou Han, Chao Shi, Tabassum Mahmud, Zeren Yang, Vladislav Esaulov, Lipeng Wan, Yong Chen, Jim Wayda, Matthew Wolf, and Mai Zheng. 2024. Revisiting erasure codes: A configuration perspective. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. 93–100.

[23] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. 2023. Perseus: A {Fail-Slow} detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 49–64.

[24] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. 2017. Proactive error prediction to improve storage system reliability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 391–402.

[25] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. 21–33.

[26] Microsoft. [n. d.]. Microsoft Bing: Search. https://www.bing.com/. Accessed: June 5, 2025.

[27] Ibomoiye Domor Mienye and Nobert Jere. 2024. A survey of decision trees: Concepts, algorithms, and applications. *IEEE access* (2024).

[28] Doug Moen. 1992. A Discipline of Error Handling.. In *USENIX Summer*.

[29] AB MySQL. 2001. MySQL reference manual.

[30] OpenStack Foundation. 2010. OpenStack Swift: Object Storage. https://docs.openstack.org/swift/latest/. Accessed: 2026.

[31] OpenStack Foundation. 2013. python-swiftclient: OpenStack Object Storage API Client Library. https://docs.openstack.org/python-swiftclient/latest/. Accessed: 2026.

[32] Oracle. [n. d.]. Oracle | Cloud Applications and Cloud Platform. https://www.oracle.com/. Accessed: June 5, 2025.

[33] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 235–248.

[34] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, William S Beebee, et al. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing.. In *Osdi*, Vol. 4. 21–21.

[35] Jeff Dean Sanjay Ghemawat. [n. d.]. LevelDB. https://github.com/google/leveldb. Accessed: June 5, 2025.

[36] Amazon Web Services. [n. d.]. Amazon S3 Object storage built to retrieve any amount of data from anywhere. https://aws.amazon.com/s3/. Accessed: June 5, 2025.

[37] Erik Stenman. 2018. Beam: a virtual machine for handling millions of messages per second (invited talk). In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 4–4.

[38] Theodore Ts'o and Inc. PowerQuest. [n. d.]. resize2fs - ext2/ext3/ext4 file system resizer. https://linux.die.net/man/8/resize2fs. Accessed: June 5, 2025.

[39] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*. 307–320.

[40] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and detecting disordered error handling with precise function pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. 2041–2058.

[41] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. 2019. Lessons and actions: What we learned from 10k {SSD-Related} storage system failures. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 961–976.

[42] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[43] Yahoo. [n. d.]. Yahoo | Mail, Weather, Search, Politics, News, Finance, Sports ... https://www.yahoo.com/. Accessed: June 5, 2025.

[44] Amirhessam Yazdi, Xing Lin, Lei Yang, and Feng Yan. 2020. SEFEE: Lightweight storage error forecasting in large-scale enterprise storage systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[45] Di Zhang, Chris Egersdoerfer, Tabassum Mahmud, Mai Zheng, and Dong Dai. 2023. Drill: Log-based anomaly detection for large-scale storage systems using source code analysis. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 189–199.

[46] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. 2023. Multi-view feature-based {SSD} failure prediction: What, when, and why. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 409–424.

[47] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 449–464.

[48] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of {SSDs} under power fault. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 271–284.

[49] Mai Zheng, Duo Zhang, and Ahmed Dajani. 2026. On fault tolerance of data storage systems: A holistic perspective. *Fault Tolerance in Modern Engineering Systems* (2026).