Assessing the Fault Tolerance of Data-Centric Applications

Maria Ramos
Supervised by João Paulo and Tânia Esteves
1st year Ph.D student at University of Minho and INESC TEC
maria.j.ramos@inescetc.pt

ABSTRACT

Data is considered one of the most valuable assets for organizations, leading to ever-increasing demands for its availability, consistency, and reliability. However, data-centric applications, such as databases, are often vulnerable to faults stemming from both internal factors, such as software bugs, and external factors, like power failures. Consequently, ensuring fault tolerance in these systems presents a critical challenge.

This Ph.D. thesis aims to research a new generation of methodologies and tools for effectively and automatically assessing the fault tolerance of complex data-centric software systems. First, addressing this challenge requires navigating a wide range of issues. Modern systems are increasingly complex and non-deterministic, making it difficult to analyze faults and reproduce failures. Second, existing fault injection tools often lack a balanced approach between exploration and targeted testing, provide limited root cause insights, and pose trade-offs between fault coverage and performance. Addressing these limitations is crucial to make vulnerability detection and resolution more practical and effective.

Finally, although there is a substantial body of research on storage fault injection, the field remains highly fragmented. This lack of systematization makes it difficult for testers to identify the most suitable techniques for their use cases or to understand how different tools can complement each other.

VLDB Workshop Reference Format:
Maria Ramos. Assessing the Fault Tolerance of Data-Centric Applications. VLDB 2025 Workshop: PhD Workshop.

1 INTRODUCTION

Data integrity refers to the accuracy, consistency, and reliability of data throughout its lifecycle—during storage, retrieval, and transmission. Data-centric systems, such as databases, are fundamentally built around maintaining data integrity, as their primary role is to store, update, and retrieve data correctly. Many of these systems, especially those implementing strong guarantees like ACID (atomicity, consistency, isolation, durability), must uphold strict standards of correctness in environments that are filled with potential faults [1].

Because modern software is composed by many components, the failure of an individual component may not imply a system-level failure unless the erroneous state becomes externally visible. The underlying cause of a failure is a fault. Faults can arise from hardware malfunctions, software bugs, or external factors such as power failures and environmental conditions. These faults can manifest in different ways, including data corruption, or data loss. Particularly concerning cases are those of silent data corruption, where systems return wrong data to users or other software components without raising any explicit error or warning [8, 19]. In order to design robust systems, it is essential to understand the nature of these faults, their causes, and their potential impact on storage systems and applications.

Among the most harmful and concerning faults are storage-related faults [8]. The expectation when data is written to storage is that it will remain unchanged, uncorrupted, and faithfully represent the application's intended state. However, this assumption is frequently challenged in real-world scenarios. Applications typically rely on several assumptions: (i) that stored data will not undergo unintended modifications; (ii) that data consistency is preserved across storage layers unless explicitly relaxed by the system's design (as in eventual consistency models); (iii) that data will persist reliably despite crashes or power failures; and (iv) that storage operations are atomic, either fully completed or not performed at all, thus avoiding partial writes.

Violations of any of these assumptions can result in subtle yet severe faults within data-centric applications. Such faults may go undetected for extended periods, further compounding their impact. As a result, ensuring data integrity in the presence of diverse and often unpredictable storage faults remains a fundamental requirement for the development of resilient and dependable systems.

1.1 Challenges

There is already a substantial body of work on storage fault injection, encompassing a wide range of tools that introduce different types of faults, employ diverse strategies for uncovering vulnerabilities in fault-tolerance mechanisms, and produce varied forms of reports from fault injection experiments [2–4, 6–9, 12, 15, 17, 21–23]. Due to this diversity, there is a pressing need for a systematic organization of this knowledge. The field's contributions are spread out, meaning that such systematization would help researchers and practitioners understand how these tools can complement one another, identify the scenarios in which each tool is most effective, help understand how these tools fit within workflows and highlight opportunities for advancing the field.

Fault injection tools face several critical challenges, which are worsened by the increasing complexity of modern systems. Modern software systems are composed of multiple interacting layers, including storage subsystems, caching mechanisms, buffering strategies, and I/O reordering techniques, all of which contribute to the difficulty of reasoning about fault tolerance and the correctness of

Proceedings of the VLDB Endowment. ISSN 2150-8097.

the implemented mechanisms [5]. These layers introduce subtle interactions that may only surface under specific conditions, making faults both harder to trigger and more complex to analyze.

Non-determinism. Non-deterministic behaviors such as thread interleaving, message reordering, and asynchronous disk operations introduce uncertainty. These sources of non-determinism complicate the task of reproducing failures, which is essential for understanding and fixing vulnerabilities exposed through fault injection [24].

Explainability. Many tools struggle not only to expose faults consistently, but also to provide meaningful insights into their root causes. Without detailed diagnostic information, developers may find it challenging to interpret fault injection results or make informed decisions about improving system robustness.

Scope and performance. Many fault injection tools impose a trade-off between fault coverage and performance. Tools that achieve high fault coverage often incur the problem of state-space explosion [7], which can discourage developers from using them in real-world development or deployment environments. This performance impact, combined with limited root-cause analysis and the intrinsic complexity of modern systems, represents a substantial barrier to the practical adoption of fault injection in day-to-day software engineering workflows.

The primary focus of this Ph.D. research is to develop novel methodologies and tools for fault injection aimed at improving the robustness and dependability of data-centric systems. This work seeks to address the key limitations identified in the current state of the art, offering practical solutions to enhance fault detection, analysis, and resilience.

2 STATE OF THE ART

In recent decades, a significant number of fault injection tools and techniques have been developed and implemented. Typically, the fault models of these tools vary, *e.g.*, they target different types of vulnerabilities. These differences often make these tools complementary, a fact acknowledged in several research studies. This suggests that many of these tools serve distinct yet interconnected purposes and can be effectively integrated.

Fault injection tools follow a structured sequence of operations that include finding, reproducing, and explaining bugs. Different tools may contribute to different stages within this pipeline, each fulfilling a unique role in the overall fault injection process. Understanding how these tools interact and complement one another not only reveals their collective strengths but also uncovers gaps in the current state of the art, offering valuable insights into areas requiring further research and development. Figure 1 illustrates a representation of a generic fault injection pipeline.

Among the presented activities , **bug finding** Figure 1-(1) is the primary focus of most fault injection tools. These tools systematically inject faults into a system to perturb its normal execution, forcing it to explore different execution paths that may reveal hidden bugs. This process can be called exploration, as the tool tries to explore the system's vulnerabilities. The effectiveness of a fault injection tool in bug finding mostly depends on its **search strategy**.

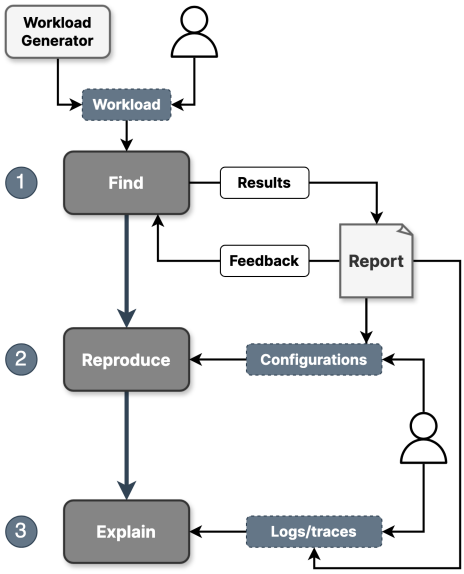

Figure 1: Overview of the fault injection pipeline, highlighting the stages of bug finding, reproduction, and explanation.

Tools employ different search strategies to determine where and when to inject faults. Some use random injection [11, 23], while others apply heuristics [13, 14] or systematic fault exploration to maximize coverage [3, 12, 18, 21]. Ideally, the optimal search strategy efficiently uncovers all vulnerabilities with minimal failed attempts or redundant tests. A common problem in exploration is the **state space explosion**, where the number of possible states grows exponentially, making it challenging to explore all possible paths [12, 15, 21, 22]. The level of automation of exploration also varies as some tools require users to specify a workload, while others automatically generate workloads and/or refine input generation dynamically to increase the likelihood of triggering bugs.

Once a bug is detected, the tool provides output that informs the user about the discovered issue. This output may include details such as the stack trace, the specific input that triggered the bug, or the system state at the time of failure. This information might be crucial for **bug reproducibility** Figure 1-(2), which involves repeating the test that led to the bug. Reproducing a bug is one of the most crucial steps in debugging. By reproducing the issue, developers can analyze system behavior and identify the root cause.

Reproducing a bug is often challenging, with developers spending a significant portion of their debugging time simply trying to replicate failures [24]. Most bugs arise from unexpected event orderings (e.g., messages, I/O) within a system, and recreating the conditions that led to the bug is often very challenging, especially in systems with complex interactions. This non-determinism that is inherent to many systems makes reproducing bugs a difficult task.

Bug reproduction is also critical in production testing to to confirm that fixes work [24]. In the process of reproducing a bug, there are two distinct approaches. The first approach involves tools that focus on finding a strategy to reproduce a specific bug [17]. These tools inject faults strategically to find a way to replicate a known

failure. Their goal is not broad vulnerability discovery but rather the precise replication of a particular issue.

The second approach includes tools that operate based on a predefined configuration to reproduce a specific bug [19, 20]. This configuration can either be manually provided by the user or automatically generated by the tool itself, particularly if the tool previously performed a bug-finding phase. In this case, the tool leverages prior knowledge of system vulnerabilities to systematically recreate the conditions that led to a given failure. One example of this type of tool is record-and-replay tools, which record all of the system's execution and replay it to reproduce the bug [10].

Once a bug can be reliably reproduced, the next step is to pinpoint its root cause. This involves identifying the exact conditions that trigger the failure and understanding the sequence of events leading to it. However, diagnosing a bug is not just about finding its source since it is equally important to explain how the root cause leads to the failure and how it propagates through the system.

To address this need, we identified **bug explanation** Figure 1-(3) as the final stage in the fault injection pipeline. This stage aims to provide a comprehensive understanding of the bug's root cause, its impact, and how it propagates through the system.

Bug explanation involves analyzing various system aspects, including interactions between components, data flow, and system state transitions. The goal is not only to identify the initial trigger but also to trace how the bug manifests across different layers of the system. This analysis helps determine the points at which the failure could have been prevented and the most effective approach for fixing the issue.

The primary focus of this Ph.D. thesis is to develop novel methodologies for fault injection aimed at improving the robustness and dependability of data-centric systems. While formal software verification is not the central objective of this thesis, we acknowledge its value in the context of fault injection. Therefore, we will consider its potential application in concrete scenarios, such as verifying the correctness of system behavior after injecting faults.

3 OBJECTIVES

The state of the art highlights three fundamental challenges, but there is still confusion regarding the properties and capabilities of existing tools, such as whether they allow bug reproducibility and what type of reports on bugs they provide.

Therefore, the first goal of this Ph.D. is to provide the first comprehensive systematization review of fault-injection tools and a novel taxonomy that will allow academia and industry to better understand how existing tools complement and the research gaps still to be addressed. Then, we will make specific contributions for the challenges highlighted in Section 1.1.

More generally, this Ph.D. aims to advance the state of the art in fault injection by developing a new generation of methodologies and tools for assessing the fault tolerance of data-centric systems. The following steps outline the main directions of this work.

The first step involves the development of an automated and comprehensive solution for fault assessment. Current tools in the literature typically struggle to strike a balance between broad exploration and targeted fault injection. They either focus on randomly injecting a large number of faults, making it hard to understand and reproduce bugs, or they do targeted fault injection for very specific types of faults, sometimes requiring a deeper knowledge of the system being tested. This Ph.D. proposes a hybrid approach that combines the strengths of both strategies. In particular, the proposed methodology will enable the **exploration of potential vulnerabilities** without prior system knowledge by employing techniques such as dynamic analysis to inspect code and execution flow. Once potential vulnerabilities are discovered, users will be able to narrow down and **reproduce** these issues through targeted fault injection. This capability is critical for enabling developers to isolate the root causes of faults and apply effective fixes.

Ensuring both soundness and completeness in this process will be a core objective. A sound methodology must accurately reproduce faults, while a complete one should detect a wide range of possible vulnerabilities. Existing tools are often restricted in the types of faults they consider, for instance, some focus solely on network-level issues, while others focus on storage problems. This thesis will build on prior work [19] by progressively introducing a broader variety of fault types, enabling more thorough coverage of potential failure scenarios.

The next focus of this research will be **controlling non determinism** during testing. As systems grow more complex, especially in distributed environments, non-deterministic behaviors such as thread interleaving, asynchronous I/O, and message reordering make it difficult to consistently reproduce and analyze faults. To address this challenge, the proposed work will explore techniques for minimizing variability across test runs. This includes strategies for controlling disk persistence, network transmission, and internal operation ordering. In distributed systems, reducing non-determinism is particularly challenging due to the need for inter-node coordination. Consequently, this work will explore advanced observability and event correlation techniques such as context propagation [16] combined with fault injection.

Another key step will be enhancing the explainability of detected vulnerabilities. Fault injection is most effective when it not only exposes errors but also provides insight into their root causes. Current tools often fall short in this regard, offering limited contextual information. This work will aim to improve root cause analysis by collecting and correlating data from multiple system layers during and after fault injection. This will involve using current observability and monitoring techniques to support fault injection and better explain bugs.

Through these steps, this research aspires to significantly improve the effectiveness of fault injection as a methodology to ensure the robustness and dependability of modern data-centric systems.

4 WORK IN PROGRESS

This research builds upon our previous work, LazyFS [19], which is already integrated into the first year of my Ph.D. studies. LazyFS is a FUSE-based fault injection framework specifically designed to simplify the debugging and reproduction of complex data durability bugs that arise during crash scenarios in systems such as databases, key-value stores, and blockchain platforms.

LazyFS has been used to multiple widely-used systems. Through this evaluation, LazyFS successfully reproduced 12 previously reported bugs and uncovered 8 new bugs. The tool operates as a

passthrough I/O layer between the System Under Test (SUT) and a persistent file system backend (e.g., ext4), intercepting and managing file system operations.

The key innovation in LazyFS lies in its control over the operating system page cache. Unlike standard Linux file systems, LazyFS does not perform background flushes. Instead, data written by the SUT is held in LazyFS's internal cache and only flushed to the backend file system when an explicit synchronization operation (*e.g.*, fsync) is issued by the SUT. This controlled flushing enables LazyFS to simulate crash scenarios involving lost or torn writes.

Users can specify fault injection policies, including the type, timing, and location of faults, allowing LazyFS to deliberately cause full or partial data loss of unflushed writes. The tool then enables developers to analyze system behavior under these fault conditions by inspecting resulting inconsistencies or errors, thus supporting a detailed investigation into crash consistency violations.

Through its ability to reproduce realistic crash-consistency faults and support detailed failure analysis, LazyFS provides a valuable foundation for advancing fault injection methodologies targeting data-centric systems. Currently, we are expanding the range of faults that LazyFS can inject and automating the process of vulnerability exploration.

ACKNOWLEDGMENTS

This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2023 (https://doi.org/10.54499/UID/50014/2023) and through the Ph.D. grant with reference 2024.02567.BD.

REFERENCES

[1] ACID Transactions: The Cornerstone of Database Integrity [n.d.]. https://www.yugabyte.com/acid/acid-transactions Accessed on 22/05/2025.

[2] Ram Alagappan and Peter Alvaro. 2022. Crash Consistency: Keeping data safe in the presence of crashes is a fundamental problem. *Queue* 20, 4 (Sept. 2022), 107–115. https://doi.org/10.1145/3561654

[3] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated Crash Vulnerabilities. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 151–167. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/alagappan

[4] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 331–346. https://doi.org/10.1145/2723372.2723711

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[6] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 83–98. https://doi.org/10.1145/2980024.2872406

[7] Wenhan Feng, Qiugen Pei, Yu Gao, Dong Wang, Wensheng Dou, Jun Wei, Zheheng Liang, and Zhenyue Long. 2024. FaultFuzz: A Coverage Guided Fault Injection Tool for Distributed Systems. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 129–133. https://doi.org/10.1145/3639478.3640036

[8] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 149–166. https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan

[9] Gregory R. Ganger and Yale N. Patt. 1994. Metadata update performance in file systems. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation* (Monterey, California) *(OSDI '94)*. USENIX Association, USA, 5–es.

[10] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 193–208.

[11] Jepsen. 2024. A framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen

[12] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 399–414.

[13] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 419–431. https://doi.org/10.1145/3173162.3177161

[14] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 114–130. https://doi.org/10.1145/3341301.3359645

[15] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. https://doi.org/10.1145/3302424.3303986

[16] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 378–393. https://doi.org/10.1145/2815400.2815415

[17] Jia Pan, Haoze Wu, Tanakorn Leesatapornwongsa, Suman Nath, and Peng Huang. 2024. Efficient Reproduction of Fault-Induced Failures in Distributed Systems with Feedback-Driven Fault Injection. In *ACM SOSP 2024*. ACM. https://www.microsoft.com/en-us/research/publication/feedback-driven-fault-injection-efficiently-reproducing-fault-induced-failures/

[18] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: on the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 433–448.

[19] Maria Ramos, João Azevedo, Kyle Kingsbury, José Pereira, Tânia Esteves, Ricardo Macedo, and João Paulo. 2024. When Amnesia Strikes: Understanding and Reproducing Data Loss Bugs with Fault Injection. *Proc. VLDB Endow.* 17, 11 (July 2024), 3017–3030. https://doi.org/10.14778/3681954.3681980

[20] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. Can Applications Recover from fsync Failures?. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 753–767. https://www.usenix.org/conference/atc20/presentation/rebello

[21] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, USA, 213–228.

[22] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA) *(OSDI '06)*. USENIX Association, USA, 10.

[23] Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. 2022. Maximizing Error Injection Realism for Chaos Engineering With System Calls. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2022), 2695–2708. https://doi.org/10.1109/TDSC.2021.3069715

[24] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, Shanghai China, 19–33. https://doi.org/10.1145/3132747.3132768