Analyzing Metastable Failures

Rebecca Isaacs
AWS
USA

Peter Alvaro
UC Santa Cruz and AWS
USA

Rupak Majumdar
MPI-SWS and AWS
Germany

Kiran-Kumar Muniswamy-Reddy
AWS
USA

Mahmoud Salamati
MPI-SWS
Germany

Sadegh Soudjani
MPI-SWS
Germany

ABSTRACT

A metastable failure is a self-sustaining congestive collapse in which a system degrades in response to a transient stressor (e.g., a load surge) but fails to recover after the stressor is removed. These rare but potentially catastrophic events are notoriously hard to diagnose and mitigate, sometimes causing prolonged outages affecting millions of users.

Ideally, we would discover susceptibility to metastable failures through testing. However, this is infeasible without first narrowing down the vast parameter space, which comprises both static configuration properties and dynamic behaviors. In this paper we propose a practical approach via an integrated ensemble of tools with varying levels of abstraction: probabilistic models drawn from queueing theory, discrete event simulators, and service emulators. Using a simple but realistic example, we show how an engineer can iteratively develop a performance model that predicts the conditions under which the service is vulnerable to metastable failures.

ACM Reference Format:

Rebecca Isaacs, Peter Alvaro, Rupak Majumdar, Kiran-Kumar Muniswamy-Reddy, Mahmoud Salamati, and Sadegh Soudjani. 2025. Analyzing Metastable Failures. In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730380

1 INTRODUCTION

Failures in distributed systems are common and software engineers build resilience by drawing from a plethora of mechanisms and strategies, including timeouts, retries, queues, caches, health checks, load balancing, hedging, load shedding, and many others. The goal is for systems to detect and mitigate failures quickly with minimal disruption to the system's availability and performance—and speaking as employees at a hyperscaler, these resilience mechanisms are effective in preventing or mitigating the vast majority of failures in the public cloud.

However, there is a class of rare failures, colloquially known as *metastable failures*, that do not recover transparently, but rather stay in a degraded state even after the triggering condition is removed. In a metastable failure, a *sustaining effect*, such as a retry storm or increased work from error handling, prevents the system from making progress on its backlog of work, which in turn exacerbates the sustaining unhealthy behavior. In the worst cases the failure propagates to other systems, resulting in congestive collapse of multiple inter-dependent services and leading to prolonged outages that can affect millions of users. Widely-reported recent examples include outages at OpenAI [1] and AWS [2].

Over the past decade, our understanding of metastable failures has improved tremendously, from folklore and anecdotes [14, 20], to definitions and characterization of the problem [6], to taxonomies, models, and reproduction of past outages [10, 12]. Yet, despite all this understanding, large-scale, cascading failures continue to plague Internet enterprises and their customers: our gains in recognition and reproduction have not yet transferred to analysis and prediction.

In this paper, we tackle the open problem of identifying where systems are vulnerable to metastable failures *before they fail.* We describe a promising approach for metastability analysis that enables the service operator to obtain a macroscopic view of safe operational characteristics and conditions under which a system can fall into a metastable state, revealing information that can be used for effective defense and mitigation techniques.

Our toolchain, shown in Figure 1, analyzes the same server configurations at different levels of abstraction. These range from a mathematical description based on queueing theory models, to simulation, emulation, and stress testing. As we

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

HOTOS 25, May 14–16, 2025, Banff, AB, Canada

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1475-7/25/05.
https://doi.org/10.1145/3713082.3730380

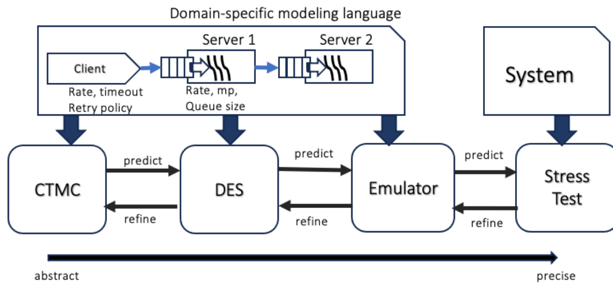


Figure 1: Tool suite for metastability analysis.

move from the fully abstract to more concrete representations, the measurements are increasingly accurate reflections of reality, but the cost of obtaining them—especially for questions about the entire configuration state space of the system—rises steeply. Importantly, the tools are tightly coupled: empirical observations from more concrete tools inform the abstract models, while predictions from abstract models direct experiments to specific parameter settings.

In the rest of this paper we will describe each tool in turn, showing the effectiveness (and limitations) of the approach using a simple, but realistic example. We believe that this research opens the door to a principled, yet pragmatic, approach to the problem of identifying when systems are vulnerable to metastable failure.

2 OVERVIEW

We focus on a simple but important setting: request-response systems with one or more request queues serviced by a thread pool. This style of architecture is typical for control plane services, which are often at the heart of significant outages precipitated by metastable failures. In future we plan to extend this work to other types of distributed systems, for example decentralized, gossip-style systems.

Figure 1 provides an overview of the set of tools, while Table 1 compares their expressiveness and costs. A domain-specific modeling language (DSL), embedded into Python, describes the system using abstractions such as thread pools, queues, requests, and retry policies. We have studied several request-response systems running in production, and have found that a surprisingly small number of core abstractions is sufficient to characterize their high-level dynamics. Business logic is replaced with a service time distribution–i.e., we only model the delay during which a thread is unavailable to do other work.

The modeling language is shared among a suite of tools, each giving a more precise semantics to the core abstractions in the language. At one extreme is a mathematical model, a *continuous time Markov chain* (CTMC), which provides an abstract *average-case* view of a system, eliding the operational details of the constructs. At the other extreme is an emulator that implements the model using a production RPC framework deployed on the same cloud infrastructure as the real system. In the middle, a discrete event simulator (DES) provides an operational model, which implements the algorithms required for the core abstractions (for example, thread pool scheduling), while abstracting away from real communication and infrastructure. Finally, the ground truth is provided by the actual service implementation and stress tests against the deployed service.

As we move from stress testing at one extreme to CTMC modeling at the other, we abstract away details that are (ideally) inessential in order to allow more efficient global reasoning across all possible system executions, while leaving the underlying common abstraction of queues, timeouts, and retries fixed. As we discuss later, initial analyses with the CTMC takes the order of seconds, the DES minutes, and the emulator several hours. On the flip side, the fidelity of the analysis to the ground truth is higher as the models get more precise.

Of course, modeling is an art form; if we abstract away important details, we may obtain efficient models that tell us nothing useful about the real system. Thus, each time we abstract from one tool to the next in the chain, we can calibrate with the concrete representation. Thus, we use more abstract tools to provide *predictions* about global behaviors, we *check* the predictions using more precise semantics, and *refine* the abstraction—and possibly also the model of the system—based on the outcomes.

In the following sections we will describe the tools and their analysis through a simple running example. Figure 2 shows the DSL listing: it consists of a server with a default queue bound of 150, a single thread serving API requests at 10 RPS and a single client that, under normal situations, makes API requests at 5 RPS, with a timeout of 5s and five retries. Our goal is to understand *when* this system can tip and thus how to tune the system's parameters to defend against it. Along the way, we discuss the predictions (from left to right in the tool suite) and refinements (from right to left), the necessity for several levels of abstraction, and their tradeoffs in metastability analysis.

3 CONTINUOUS-TIME MARKOV CHAINS

We start with the question: under what parameterizations is the running example vulnerable to a metastable failure? For a service operator configuring a system, understanding such global operational characteristics is important, but load testing a service across a broad swathe of parameters is simply too expensive.

Tool	Expressiveness	Analysis	Cost
Visualization	Queueing and retry state space dynamics	Summarize state evolution trends	Very low (ms)
CTMC	Mathematical, macroscopic description	Predict average, whole-system performance	Low (s)
Simulation	Implementations of core abstractions	Confirm predictions, deep-dive observations	Medium (mins)
Emulation	Production infra, no business logic	Validate and verify model	High (hours)
Real system	Perfect fidelity	Stress test for complete confidence	Very high (hours+)

Table 1: Comparison of tools.

```
1 def program():
2   api = { 'insert': Work(10, [],) }
3   server = Server('simple', api,
4                   qsize=150, thread_pool=1)
5   src = Source('client', 'insert', rate=5,
6                   timeout=5, retries=5)
7   p = Program('SimpleService')
8   return p.add_server(server).add_source(src).
      connect('client', 'simple')
```

Figure 2: A request-response example in our DSL.

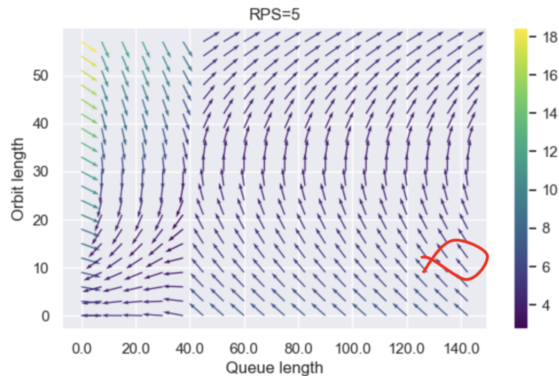


Figure 3: Visualization of queueing and retry dynamics. The overlaid circle indicates where the running example metastable simulation (described in Section 4) ends up after the trigger subsides.

We jump to how the CTMC model provides an answer. Figure 3 shows a visualization of the queueing dynamics of the system, derived automatically from the model. On the x-axis we show the current queue length, and on the y-axis the number of requests waiting to be retried, known as the *orbit* (think of this as capturing the potential work amplification in the system). The behavior of the system is probabilistic: the arrow at each (x, y) coordinate represents the *most likely* next transition, both in terms of direction and, in the heat map color, the strength of the probabilistic choice (lighter colors indicate higher probabilities). What is notable is that there is an obvious inflection point at which the system looks most likely to move towards a long queue and large numbers of retries in flight and thereafter get "stuck". In the figure, which models the running example with a load of 5RPS, this occurs at around a queue length of 40 with at least 30 requests retrying: above and to the right of this point, the likely outcome of trajectories is that they enter the self-sustaining feedback loop of a metastable failure. By contrast, when in the lower left region of the state space, the likely outcomes will tend towards an empty queue and thus safety.

For a service operator, this visual summary is immediately useful. They can quickly see when queue lengths and retry rates put their service into a vulnerable state, and moreover, because the visualization takes only milliseconds to generate, can interactively explore how the configuration space impacts queueing behavior and the probability of falling into a metastable failure state.

The visualization is generated from a CTMC, which is a probabilistic model of the system. CTMCs are a standard modeling framework in queueing and performance analysis [3, 10, 11]. Intuitively, the state of the CTMC represents an abstraction of the system state—the number of requests being served or queued, and the number of requests being retried—and the transitions capture probabilities of transitions between states. Our visualization exploits the direct 2D nature of the states per queue in the system and the sparsity of the underlying transitions; even for large systems, these visualizations can be generated within a few hundred milliseconds[1].

In addition, the CTMC can answer *quantitative* questions, such as the average queue length at steady state (the stationary distribution) as well as the expected recovery time to the average queue length, starting from a state when the queue is full. These analyses are based on linear algebra, and reasonably fast (for the running example, these calculations finish within a few minutes.) In fact, we can provide a mathematically precise definition of metastability based on the quantitative analysis: roughly, a CTMC is metastable if it has

[1]Our CTMC implementation, including generation of the visualizations, is available from http://gihub.com/mpi-sws-rse/metafor.

two or more disjoint subsets of states, each reachable from all the others, such that the smallest expected time to reach from one subset to another is much larger than the largest expected time to reach any one subset from any state. The system "gets stuck" in a subset for a time scale much larger than the natural time scale.

The main limitation of the CTMC is the relatively coarse abstractions that are inherent in the modeling. For example, all arrival and service rates are modeled as exponential distributions. Instead of maintaining precise retry counts and individual timers to detect timeouts, the model globs them into a probability distribution with the same expected behavior (but whose distribution can be quite different!). The precision of these abstractions get worse as the work amplification increases, so quantitative predictions from the model can deviate considerably from what we observe in simulation and emulation.

Learning more precise CTMC representations from data is an important open problem that we discuss further in Section 7. Nevertheless, we find the whole-system perspective of the CTMC invaluable for quickly directing attention at the inflection points of the configuration space. The low overhead of this analysis enables finding regions prone to metastability using multi-dimensional parameter sweeps that would be infeasible otherwise, even offline using simulation. Once we have an imprecise picture of the vulnerable regions, we calibrate the quantitative CTMC predictions with discrete event simulation, as we describe next.

4 SIMULATION

Discrete event simulation (DES) in effect replays the system through a randomly chosen trajectory in the CTMC state space, but with a more precise implementation of the core abstractions. Our implementation of the DES has a core event-driven loop [13] and operational implementations for constructs in the DSL based on events.

We find that DES inhabits a sweet spot between the mathematical modeling of the CTMC and the messiness of actual service infrastructure. Not only does it validate the model's predictions, but it also provides a white box analysis of the system. We can capture any metric we wish and even perform (local) distributed tracing, enabling deep inspection of what is happening in the system, at the granularity of a single request, at any point in its execution. This is valuable not only for understanding complex queueing dynamics, but also for improving intuitions about metastable failure, which we have found is often an "aha" moment for operators of production services.

Figures 4a and 4b show the availability characteristics for a simulation using the running example with two different queue bounds. The graphs show the timeseries for tail latency in simulation runs in which the client first sends a baseline load of 5 RPS for 5 (simulated) minutes (recall that nominal capacity of the server is 10 RPS), then overloads the system with 15 RPS for 10 minutes before dropping back to 5 RPS for the remaining 30 minutes of the experiment. From the CTMC, we know the first system is unlikely to recover, and the second should recover. Simulation results confirm this—the red circle overlaid on Figure 3 shows where in the orbit/queue state space the simulation with a queue bound of 150 gets stuck in the third phase of the experiment.

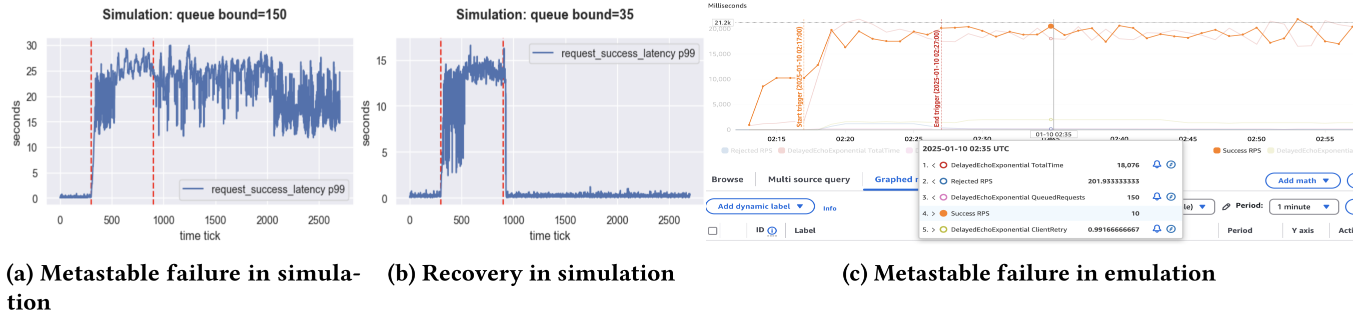
5 EMULATION

Real-world factors ignored by our models and simulations include processor-level resource contention, thread scheduling, the behavior of a service's load balancer, and many others. As such, there are no guarantees that factors extraneous to our model will not play a part in whether the service is prone to metastable failure. Emulation helps to compensate for this drawback: firstly by validating predictions and confirming that the system behaves comparably to the simulator, and secondly by providing empirical observations that we can feed back into the models to improve their fidelity.

To implement the emulator, we use a bare-bones deployment of a service running over a widely-deployed internal RPC framework. The service does no actual work when receiving a request but rather sleeps for a period of time drawn from a configurable distribution. As with the simulator, we are able to use both synthetic distributions (e.g. exponential) and those constructed from production metrics. A high-volume load generator acts as the client.

The Amazon CloudWatch screenshot in Figure 4c shows the 99th percentile latency of an emulated reproduction of the simulated failure shown in Figure 4a, with a queue bound of 150. The full results from the emulator are broadly comparable to those of the simulator, although one interesting divergence, which we are currently debugging, is that the fraction of requests that are retries following the trigger phase is 20% higher in the emulator compared to the simulator. This emphasizes the necessity of emulation to confirm the accuracy of results from the simulator.

We have heard anecdotally that the experimental reproductions of metastability reported in [12] involved some heroic work to induce, and we posit that at least part of the challenge was a lack of visibility into the complicated dynamics underlying the failures. In contrast, with the help of CTMC modeling and DES, we found a configuration that causes a comparable metastable failure with little effort beyond writing Python scripts to perform parameter sweeps over the core abstractions.



(a) **Metastable failure in simulation**

(b) **Recovery in simulation**

(c) **Metastable failure in emulation**

Figure 4: 99th percentile latency in the running example: (a) and (b) simulation for queue bounds 150 and 35, and (c) emulation. The dashed vertical lines indicate where the trigger load surge of 15RPS started and stopped.

6 TESTING

The highest fidelity, but most expensive, part of the tool chain is running stress tests on the real system. Such tests run slowly (i.e., in real time), use real servers, and require humans to write scripts, follow deployment procedures, ensure they have the necessary credentials, clean up afterwards, and so on. First inducing metastable failures in the emulator gives confidence that we have identified the right point in the (vast) configuration space before investing in the tests.

For testing against the real system, we have learned from service operators that simply being able to induce a single metastable failure on demand is valuable, without necessarily exploring the parameter space. Therefore our current offering outputs a single instance of configuration and load settings that our tools predict will cause the service to fail. Engineers then enter these values into their standard scripts for load testing. In future, we envisage creating more sophisticated outputs, such as automatically generated stress tests as part of continuous testing pipelines.

We note that from the perspective of a service operator, the stress testing step has by far the most credible and actionable outputs and none of the modeling, simulation, or emulation results have much practical value without confirmation against the actual system.

7 DISCUSSION

In this section we discuss how our approach generalizes beyond the simple running example presented above. We then describe how the tools in our pipeline interact in order to prune away costly executions or to improve the fidelity of an abstraction.

Applicability. For ease of exposition our running example is simple, which raises the question about the broad applicability of our approach. The example models a queuing system on a single host, with one thread to service requests, and it involves a specific type of sustaining effect—a retry storm. Other types of metastable failure, such as the slow error handling or link imbalance examples described in [6], are not explicitly properties of queuing systems, but we have found that they can generally be modeled as such with an unbounded or singleton-bound queue. For example, slow error handling is captured by changing the service time distribution for a fraction of arriving requests. Additional resource usage constraints can be modeled along with "scheduling" a dequeued request to a worker thread, although we acknowledge that modeling resource usage can be arbitrarily complex, for example to capture heap usage and GC behavior.

In practice, we have found that fairly simple models have been sufficient to study a range of failure types in a variety of AWS services. An important realization for us is that cascading collapse can be captured by straightforward composition of multiple models. To date, we have simulated up to three dependent services, with the scale only limited by the additional engineering work required.

Calibration. When the stages of the pipeline are calibrated, we traverse it in the order presented above, using the more abstract components to prune away unnecessary executions of the more concrete. Hypotheses formulated by querying or visualizing the CTMC (e.g., whether certain system configurations are stable in the face of load bursts, how long we can expect a collapsed system to recover after its load returns to normal, or to what degree traffic should be throttled to allow a system to recover in a shorter, fixed period of time) are validated in simulation via familiar metrics such as end-to-end request latency and goodput. Parameter sweeps that generated many thousands of CTMC analyses can be tested with a few dozen simulator runs. With a calibrated pipeline, the configurations smoke-tested by the simulator can be explored via emulation and ultimately stress testing, in every case incurring a higher execution cost for greater precision, but controlling this cost by filtering the number of required executions.

This strategy works if each abstract component produces behaviors close to its concrete neighbor. Emulations must

produce similar timeseries to the concrete stress tests they mock, agreeing on key service metrics. The same is true for DES simulations of emulated tests. When the outputs do not agree, something is missing from the abstract model and it must be further calibrated. Anecdotally, corrections to the emulator and simulator abstractions often involve modeling *more*, and some manual effort is inevitable.

The most challenging piece of calibration is to capture more complex real-world behaviors in the CTMC model. The mapping between the corresponding states of simulator and CTMC is more abstract than the almost 1-1 mapping of emulator to simulator state, and so are the required corrective mechanisms. A CTMC refines its corresponding DES model if the corresponding DES states ((q, o) for queue length q and o requests "in orbit") have similar transition probabilities. Note that there are many more states in the DES, where transitions depend on history, unlike in a Markov system, and so this is unfortunately harder to check than metrics in timeseries but potentially easier to correct.

At the time of writing, using the DES machinery to calibrate the CTMC remains a work in progress. A promising candidate is using the DES—each execution of which conceptually samples many times from the underlying distributions modeled by its corresponding CTMC—as a platform in which to perform Monte Carlo exploration. By executing many stochastic "short walks" through the DES state space, we can provide direct estimates of the transition probabilities in the corresponding CTMC. Our hope is that in many cases discrepancies between the DES and CTMC can be addressed by adjusting the CTMC's probability distributions to match those estimated via Monte Carlo simulation. As we extend all of the components in the pipeline to model more complex flow control policies, we will learn more about the limitations of this calibration approach, and may find that we need to explicitly model more elements (e.g., admission control policies) in the CTMC itself.

8 RELATED WORK

Metastable failures are an important source of outages, but there is little existing work on *finding* or *mitigating* them using testing or formal methods. The large parameter space makes it difficult even to reproduce such failures [12] and most existing work on fuzzing or model checking systems do not discuss finding metastable failures [7, 16, 19, 21]. In our experience, it is difficult to find metastable failures through testing alone because it requires exploring both the space of configuration parameters as well as the space of (random) request arrivals and their processing.

Metastability is a well-studied topic in control theory and dynamical systems [4, 9]; we show how the mathematical notions translate to the analysis of software systems.

Feedback and control of (stochastic) systems have been studied in the analysis of congestion in network protocols [5] and in resource management in operations [8], but the application of these techniques to metastable failures has not been explored.

Understanding the behavior of queueing networks is a core area in systems performance analysis [3, 11, 13]. Our work extends this literature, both in studying metastability in abstract mathematical models and in linking analysis results from mathematical models to real systems behaviors. A key insight is that metastability is inherently different from stability or instability studied in queueing systems: as our example shows, a stable queueing system can still exhibit metastable behaviors. Our work extends recent progress in CTMC models to study metastable failures [10], both in the mathematical modeling and in providing an ensemble of tools bridging the gap between modeling and deployment.

A different line of work [15, 17, 18] studies bugs in software that can result in bad retry behaviors, such as loops in retry policies, retries without timeouts, or unboundedly many retries. While the presence of these bugs would lead to collapses, our focus in this paper is to study metastable failures that can occur *even if* there are no explicit bugs in the code, purely through the macroscopic behavior of feedback in the overall system.

9 CONCLUSION

Research progress towards understanding metastable failures has not yet translated to better resilience against them. In this paper, we report concrete *methodological* progress towards predicting, mitigating, and ultimately preventing such events. From an abstract visualization of the safe operating range of a service and its boundaries, to concrete stress tests that confirm those boundaries, our toolchain allows us to extract predictions inexpensively from a simplified model and use them to drive our search through the space of more costly experiments. These experiments in turn provide ground truth that improves our models and the accuracy of future predictions: a virtuous cycle.

The explicit goal of this work is to rule out the possibility of metastable failures in any of the services managed by our hyperscaler; we present first steps down the path to this goal. We are already applying our methodology to understand the potential "tipping points" at which existing control plane services may become metastable, and providing recommendations to service teams about how to avoid vulnerable configurations. Our next steps are to ensure accurate prediction of triggers and recovery times, and to propose actionable interventions that can reduce recovery time.

REFERENCES

[1] API, ChatGPT & Sora facing issues. https://status.openai.com/incidents/ctrsv3lwd797.

[2] Summary of the Amazon Kinesis Data Streams service event in Northern Virginia (US-EAST-1) region. https://aws.amazon.com/message/073024/.

[3] Jesús R. Artalejo and Antonio Gómez-Corral. *Retrial queueing systems: A computational approach.* Springer, 2008.

[4] Anton Bovier, Michael Eckhoff, Veronique Gayrard, and Markus Klein. Metastability and low lying spectra in reversible Markov chains. *Commun. Math. Phys.*, (228):219–255, 2002.

[5] Bob Braden, David D. Clark, Jon Crowcroft, Bruce S. Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry L. Peterson, K. K. Ramakrishnan, Scott Shenker, John Wroclawski, and Lixia Zhang. Recommendations on queue management and congestion avoidance in the internet. *RFC*, 2309:1–17, 1998.

[6] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In Sebastian Angel, Baris Kasikci, and Eddie Kohler, editors, *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, pages 221–227. ACM, 2021.

[7] Constantin Enea, Dimitra Giannakopoulou, Michalis Kokologiannakis, and Rupak Majumdar. Model checking distributed protocols in must. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1900–1927, 2024.

[8] Tim Falzone and Ben Treynor Sloss. Using STAMP to improve resilience in Google production systems. In *;login:*, 2024.

[9] M.I. Freidlin and A.D. Wentzell. *Random perturbations of dynamical systems.* Springer, 1984.

[10] Farzad Habibi, Tania Lorido-Botran, Ahmad Showail, Daniel C. Sturman, and Faisal Nawab. MSF-model: Queuing-based analysis and prediction of metastable failures in replicated storage systems. In *43rd International Symposium on Reliable Distributed Systems (SRDS 2024)*, October 2024.

[11] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press, 2013.

[12] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[13] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, 1991.

[14] Ben Maurer. Fail at scale: Reliability in the face of rapid change. *Queue*, 13(8):30–46, September 2015.

[15] Shangshu Qian, Wen Fan, Lin Tan, and Yongle Zhang. Vicious cycles in distributed software systems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 91–103. IEEE, 2023.

[16] Maria Ramos, João Azevedo, Kyle Kingsbury, José Pereira, Tânia Esteves, Ricardo Macedo, and João Paulo. When amnesia strikes: Understanding and reproducing data loss bugs with fault injection. *Proc. VLDB Endow.*, 17(11):3017–3030, 2024.

[17] Utsav Sethi, Haochen Pan, Shan Lu, Madanlal Musuvathi, and Suman Nath. Cancellation in systems: An empirical study of task cancellation patterns and failures. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 127–141. USENIX Association, 2022.

[18] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. If at first you don't succeed, try, try, again...? Insights and LLM-informed tooling for detecting retry bugs in software systems. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, pages 63–78. ACM, 2024.

[19] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. Fail through the cracks: Cross-system interaction failures in modern cloud systems. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 433–451. ACM, 2023.

[20] Robbert van Renesse, Rodrigo Rodrigues, Mike Spreitzer, Christopher Stewart, Doug Terry, and Franco Travostino. Challenges facing tomorrow's datacenter: summary of the LADiS workshop. In Eliezer Dekel and Gregory V. Chockler, editors, *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS '08, Yorktown Heights, New York, USA, September 15-17, 2008*, pages 1:1–1:7. ACM, 2008.

[21] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 249–265. USENIX Association, 2014.