Characterizing Metastable Faults and Failures

Ali Farahbakhsh[†], Qingjie Lu[*], Lorenzo Alvisi[†], Andreas Haeberlen[*], Robbert Van Renesse[†]

[†]Cornell University, [*]University of Pennsylvania

Abstract

Metastable failures are hard to detect, prevent, and mitigate. During a metastable failure, a system exhibits self-sustaining bad behavior even in the absence of adversarial conditions. Prior work focuses on symptoms and has portrayed metastable failures as instances of self-sustaining overload. This characterization leaves the underlying failure causes and dynamics unknown, and does not account for metastable failures that do not manifest as overload.

We present the first causal characterization of metastable failures by identifying their origin in *metastable faults*, *i.e.*, structural destabilizing cycles of interaction among systems components that, in isolation, are stabilizing. Metastable failures arise when scheduling decisions let these destabilizing interactions gain the upper hand over the individual components' stabilizing tendencies. We then derive a methodology to predict metastable failures, and to build metastable-fault-tolerant (MFT) systems. We apply our methodology to three case studies, showcasing the generality of our results.

ACM Reference Format:

Ali Farahbakhsh[†], Qingjie Lu[*], Lorenzo Alvisi[†], Andreas Haeberlen[*], Robbert Van Renesse[†], [†]Cornell University, [*]University of Pennsylvania . 2026. Characterizing Metastable Faults and Failures. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Metastable failures [1, 3, 9–11, 16, 29, 30, 38, 44] have drawn significant attention in the systems community as a particularly challenging failure mode—one in which a system exhibits *self-sustaining bad behavior after a finite shock* [14, 17, 18, 27, 33, 35, 36]. For instance, a temporary surge in client load might lead to persistently low goodput even after the load goes back to normal. The system does not control the shock; however, the ensuing bad behavior is due solely to how the system works. Such failures are costly [35], hard to detect or predict [16], and difficult to mitigate [1, 9–11], making it critical to understand their underlying causes.

The prevailing characterization of metastable failures is mostly *phenomenological*: it starts from symptoms—high latency and low goodput under normal conditions—and equates metastability with self-sustaining overload [17, 35]. It then identifies the culprit as either an internal load amplification or capacity degradation [35]. By focusing on symptoms, this approach fails to fully explain the underlying causes that give rise to metastable failures, leaving us vulnerable in two ways: (*i*) when these failures occur, we lack targeted remedies—reports repeatedly show that only drastic measures such as restarting the system or disabling communication can halt a live failure [1, 3, 9–11]; and (*ii*) we risk overlooking metastable failures that do not manifest with the same symptoms.

To move beyond these limitations, we return to a fundamental principle of fault tolerance: *every failure stems from a fault.* We present the first analytical characterization of metastable failures that reveals the structural faults that cause them—henceforth called metastable faults. *Metastable faults are sins of composition*: they capture circular destabilizing interactions among individually stabilizing components. Each component responds to a shock by trying to stabilize locally, but in doing so destabilizes others, coupling them in perpetual destabilization and producing self-sustaining bad behavior. This characterization also explains the usual symptoms of high latency and low goodput: resources that in the stable state would be spent in useful work are instead continuously wasted in futile attempts to stabilize.

Building on our experience studying various metastable failures, we outline a methodology for predicting such failures and designing *metastable-fault-tolerant* (MFT) systems. Our methodology revolves around a proof outline for showing that a system is MFT, *i.e.*, that the system avoids metastable failures despite harboring a metastable fault. We also introduce a domain-specific language (DSL), Nyx, to (*i*) reproduce said failures *in vitro* and (*ii*) assist designers in applying the methodology. Nyx comes with a toolkit including an interpreter and a tool to visualize metastable failures.

Our methodology relies on a key insight: metastable failures emerge when poor scheduling favors the fault over stabilizing interactions. In the absence of stabilizing interactions, perpetual destabilization can be trivially inevitable. Metastable failures occur instead when the components *can* help each other stabilize, but any stabilization is interrupted by some destabilizing event. The reason is a fatal coupling between destabilization and poor scheduling: the scheduler favors the fault's destabilizing tendencies, pushing the system away from stability, creating in turn more destabilizing interactions, prompting the system to make more bad scheduling decisions—*ad infinitum.*

Based on this insight, our methodology augments compositions of stabilizing components with scheduling decisions that make them MFT. These decisions must: (*i*) schedule destabilizing interactions tentatively, and (*ii*) defer destabilizing interactions until stabilizing ones have achieved

global stability. The latter ensures eventual global stability; the former ensures that, once achieved, stability is never lost.

We prove metastable fault tolerance with respect to specific adversaries. Faults are notoriously difficult to locate [16, 38], and often require post-mortem analysis. However, these faults are triggered by an adversary. Therefore, proving stabilization despite an adversary imposing shocks on the system establishes tolerance against all faults that the adversary can trigger—a single proof covers a wide range of faults. We envision research on metastable faults and failures to resemble security research: new adversaries will expose new vulnerabilities.

We have applied our methodology to three case studies: (*i*) the retry storm [1, 8–11], which serves as the running example for our characterization, (*ii*) a novel case study from a major commercial gaming platform with hundreds of millions of users, where metastability manifests as oscillation, and (*iii*) the look-aside cache incident [17, 35], used to demonstrate the generality of our approach.

For the second case study, we use an internal post-mortem of an incident involving a cluster manager in one of the platform's data centers. We design a new cluster manager and prove it MFT. The fault in this incident is subtle and hard to locate without post-mortem insight, yet we show that fault tolerance is achievable without prior knowledge of the fault. The third case study instead illustrates that sometimes one can simply remove the fault, thereby eliminating also the failure.

Our methodology is a modest first step towards developing and deploying MFT software in production. Metastable faults span the entire stack [16, 38], and the vast scale of production systems makes locating them even more challenging. As a result, while the community has developed robust toolboxes to prevent, detect, and mitigate other modes of failure, *e.g.*, deadlocks [21, 34, 48], we lack an equivalent toolbox for metastable failures. Our characterization lays the foundation for such a pursuit by identifying the underlying principles, and our methodology demonstrates the practical utility of this characterization.

In short, we make the following contributions:

- We introduce metastable faults, and show how together with poor scheduling they lead to metastable failures;
- we present a methodology for predicting metastable failures and designing MFT systems; and
- we introduce Nyx, a DSL to reproduce metastable failures in vitro.

The paper is structured as follows. We first use the retry storm incident as a motivating running example (§2) to explain, given a system model (§3), our formal characterization of metastable faults (§4) and failures (§5). We then present our methodology (§6), and apply it to the privately reported incident (§7). Finally, we discuss related work (§8) and present our conclusions (§9). Space constraints prevent us from including the discussion of the look-aside cache incident here; it is provided in the supplementary materials (Section B).

2 Background and Motivation

Among reported metastability incidents, one stands out: the *retry storm*. It has been observed repeatedly in practice [1, 8–11], and much of the literature focuses on retry storms [33, 35, 36]. This makes it an ideal example to (*i*) illustrate the limits of today's phenomenological approach and (*ii*) serve as a running example in our analytical characterization (§4-5).

We start with a brief primer on Nyx (§2.1). We then instantiate the retry storm using Nyx and review it (§2.2). Finally, we argue that the phenomenological approach, while helpful, fails to fully explain metastable failures (§2.3).

2.1 Nyx Primer

Nyx is an imperative language with a small set of abstractions that repeatedly surface when studying metastability incidents: (*i*) *agents*—entities in a distributed execution that send/receive messages and update local state; (*ii*) input/output *queues* for communication; and (*iii*) internal *resources* that agents use to schedule state transitions.

A Nyx agent defines a set of tasks, similar to threads, plus a main task scheduling and executing them based on resource availability. Nyx interprets code for a group of agents and delivers requests to them during execution. Executions proceed in lock-step: at each step, every agent runs its main task once.

2.2 The Retry Storm

Consider a system with a fixed service capacity serving client requests, and imagine a sudden surge in client demand temporarily exceeding this capacity. Such a shock leads to congestion and increased latency, which in turn triggers client retries. Persist this long enough and the volume of retries grows so large that, even after client demand returns below capacity, congestion prevails. The retry loop sustains overload and high latency, creating a self-perpetuating cycle of congestion and retries.

Instantiation. Figure 1 illustrates a Nyx model of a system prone to a retry storm. The system has two agents: *server* and *retrier*.

The server has 35 units of a resource of type `CPU`[1], and two tasks: `serve` and `main`. `serve` consumes `CPU` resources to process requests from input queue `inq`—declared implicitly for every agent—and sends acknowledgments to the retrier; it abstracts away service logic. `main` schedules `serve` using a `soft` policy, allowing execution until `CPU` is exhausted. Nyx handles resource accounting; `serve` consumes one unit of `CPU` per every request, and when `CPU` is depleted, `serve` pauses until the next step, when `CPU` is replenished. For simplicity, we assume that `inq` has infinite capacity.

[1] This name is evocative, not tied to an actual CPU.

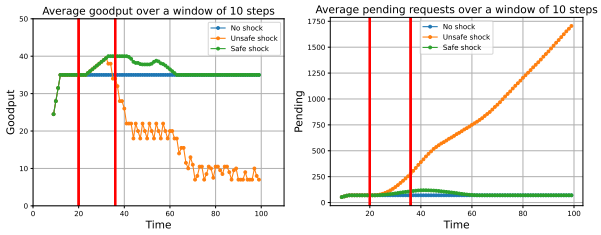
```
RETRIER:
 Task manage:
      while not inq.isEmpty():
          req ← inq.get()
          if req is "client":
              send(req, server)
              pending ← pending ∪ {req}
          else if req is "ack":
              pending ← pending \ {req}
 Task retry:
      for req in pending:
          timers[req] += 1
          if timers[req] == 4:
              send(retry, server)
              timers[req] = 0
 Task main:
      execute(manage)
      execute(retry)

SERVER:
 init resources = <CPU: 35>
 Task serve consumes <CPU>:
       while not inq.isEmpty():
           req ← inq.get()
           send(ack, retrier)
 Task main:
       execute(serve, soft)
```

Figure 1. Nyx expression for the retrier and the server. The client is not shown for lack of space. The server's finite capacity is captured by its consumption of `CPU` and the `soft` execution policy.

The retrier consists of three tasks: `manage`, `retry`, and `main`. `manage` handles incoming requests—adds `client` requests to state variable `pending`, forwards them to the server, and removes them upon receiving an `ack`. `retry` implements a retry loop. For each pending request, it increments its timer: upon exceeding a threshold (every four steps), it sends a retry to the server and resets the timer. `main` schedules both `manage` and `retry` without resource constraints. The retrier logically centralizes client behavior, behaving similar to multiple clients acting in concert. We assume an external agent (omitted for brevity) that continuously sends `client` requests to the retrier in an open loop.

Demonstration. Figure 2 shows the results of running our retry storm instantiation. The server processes $s = 35$ requests per timestep under a nominal client load of $r = 30$, with retrier timeout $T = 4$ timesteps. We analyze three scenarios: (*i*) *No shock* – the system operates normally; (*ii*) *Safe shock* – a transient surge in client load that does not destabilize the system; and (*iii*) *Unsafe shock*, a surge that triggers self-sustaining congestion. The figure reports two metrics: pending requests in the retrier (a proxy for latency), and acks received by the retrier (a proxy for goodput).

Under an unsafe shock, the backlog of pending requests grows unbounded, and goodput collapses. A safe shock causes only temporary disruption and even a short-lived goodput boost due to the elevated client load.



a Goodput. **b** Pending requests.

Figure 2. Nyx instantiation of the retry storm for $s = 35$, $r = 30$, and $T = 4$. The red lines indicate when the surge in client load starts and ends.

Review of the Current Approach. The prevailing characterization of metastability [35] explains the retry storm as a case of *retry-driven workload amplification*. It starts from how metastable failures usually manifest—high latency and low goodput under otherwise normal operating conditions—and puts the blame on either an internal amplification of load or a degradation of service capacity. These are the self-sustaining effects of a feedback loop that perpetuates the metastable failure. The shock temporarily amplifies workload or degrades capacity, long enough for the self-sustaining effects to take hold. This framework thus defines four classes of metastable failures, based on whether the shock and sustaining effect amplify workload or degrade capacity.

2.3 Shortcomings of the Current Approach

The prevailing approach focuses largely on observable symptoms, and remains tied to categories like overload and congestion, ultimately equating metastability with "self-sustaining congestive collapse" [36]. It suffers from three shortcomings.

(1) No Causal Insight. It ignores the underlying causes and internal dynamics of metastable failures. If metastability were the flu, this approach treats the fever, not the virus.

In the retry storm (Figure 1), once a shock persists long enough, retries flood the server's queue, crowding out original requests. The server wastes cycles acknowledging retries for requests already served, effectively reducing its capacity and amplifying the retrier's workload. The root cause is not just the retry feedback loop and its amplification of load; it is rather a circular interplay between workload amplification and capacity degradation, shaped by (*i*) the agents' functional composition and (*ii*) their scheduling policies, which contribute to workload amplification or capacity degradation.

(2) Inadequate Remedies. Without causal insight, attempted remedies are often suboptimal. Shedding client load is the standard response, yet our analysis shows the real issue lies in scheduling. The retrier retries aggressively every T timesteps,[2] and switching to exponential backoff for retries eliminates the failure. Revisiting the server's passive

[2] Finite retries do not eliminate metastability; we assume indefinite retries for simplicity.

scheduler is even better: prioritizing original requests over retries (*e.g.*, by maintaining separate queues) can eliminate metastability without the latency penalty of backoff.

(3) Narrow Coverage. It lacks coverage for instances of metastability that do not manifest as high latency and low goodput. We observed several such incidents; for example, metastability can manifest as oscillations in the number of active servers in a distributed system (§7). These diverse manifestations motivated us to move beyond the current characterization and develop the one presented in §4-§5.

3 Model Sketch

To reason about how components destabilize each other, we need a model that captures state changes, action effects, and how components read and write shared state. We opt for intuition, and defer a formal model to Section A.1 of the supplementary materials.

A system $S = (\Sigma, \mathcal{A})$ is a state machine with a set of states Σ, actions $\mathcal{A}$, and variables. States are valuations of variables. S interacts with an *environment* that, by taking actions of its own, can modify the values of some subset of the system's variables. Environment actions are arbitrary—the system does not control the environment's behavior. We denote state transitions of the system as $s \rightarrow s'$, where s and s' are system states.

A predicate P is a subset of the system's states, and an environment predicate E is a predicate that involves only the variables that the environment can touch. Every execution of the system is a trace of states connected via state transitions, where at each transition at least one of the system and the environment take actions. For every execution σ of the system, all suffixes of σ are also executions of the system. Given some environment predicate E and system states s and s', a transition $s \rightarrow s'$ is an E-step if s and s' satisfy E. If during a transition $s \rightarrow s'$ only the system takes an action, say α, we further label the transition as $s \rightarrow^{\alpha} s'$.

If a system assigns a value to a variable following an action, we say the system *writes to* the variable via the action, establishing a writes-to relation between them; if a system reads the value of a state variable with an action, we say that the system *reads from* the variable via the action.

Given two systems S_1 and S_2, their composition $S_1 \parallel S_2$ is a system with the Cartesian product of S_1 and S_2's states as its state space. As for its actions, the composition takes an action when at least one of S_1 and S_2 take an action. If, upon composition, S_1 becomes responsible for writing to some variable of S_2 that was previously written to by the environment, the environment stops writing to that variable in the composition. We generalize this definition to compositions of more than two systems, and denote the composition of the n systems $\{S_i\}_{0 \le i < n}$ with $\|_{0 \le i < n} S_i$. Whenever the composition takes an action during which each component S_i takes some action α_i and the environment takes some action e, the transition is serializable [15, 42], *i.e.*, the resulting state is equivalent to that resulting after *some* serial execution of the actions $\{\alpha_i\}_{0 \le i < n}$ and e.

Given a composition of n systems $\{S_i\}_{0 \le i < n}$, we lift the writes-to relation between systems and variables to a writes-to relation between systems. If a system S_i, via some action α_i, writes to a state variable of S_i that system S_j reads from, or directly to a state variable of S_j that S_j reads from, we say that S_i writes to S_j via α_i. This relation induces a *composition blueprint*: a directed graph whose vertices are systems and whose edges represent writes-to relations between them. If S_i writes to S_j, the edge is from S_i to S_j. For a system S_i, we call the set of all systems that write to it its *writing neighbors*, and denote it with W_i. Similarly, we call the set of all systems that S_i writes to as its *reading neighbors*, and denote it with R_i.

To facilitate our proofs, which entail liveness assertions [13], we assume that in every execution of a composition S (*i*) each component takes actions infinitely often, and (*ii*) during each transition at least one writing neighbor of every component takes an action.

4 Metastable Faults

Dissecting the retry storm (§2.2) reveals two distinct contributors: the composition structure and the scheduler. The former enables components to destabilize each other, while the latter repeatedly schedules for execution the destabilizing interactions. We observe this separation in every incident we examined, which motivates us to characterize metastability by distinguishing metastable *faults* from metastable *failures*. We focus on faults in this section; we defer the discussion of failures to §5.

Faults are structural vulnerabilities arising from compositions and capture the propensity for *mutual destabilization* among *individually stabilizing* components. The composition of two components harbors a metastable fault if, in reaction to a shock, the locally stabilizing actions of each component destabilizes the other. Understanding metastable faults hinges on two questions: What are stabilizing systems? And are all faulty compositions of these systems about metastability? To answer these questions, we present (*i*) a notion of stabilization (§4.1), and (*ii*) a notion of compatibility (§4.2) that rules out trivially faulty compositions of stabilizing systems. We then define metastable faults (§4.3). We illustrate these notions using the retry storm incident (§2.2) as an informal yet intuitive running example.

4.1 Stabilizing Systems

Consider a cluster manager tasked with maintaining a fixed number of active servers under a crash failure model. A sufficiently strong shock, *i.e.*, several workers crashing, can drive the system into a state where the cluster manager's guarantee no longer holds. Correctness requires the cluster manager to eventually reach a set of states with enough active workers and to stabilize there, regardless of the shock's severity.

Such systems are known as self-stabilizing systems [20, 23]. Starting from an arbitrary state after a shock, a self-stabilizing system converges to a set of good states and remains there provided that there are no subsequent shocks. Inspired by this notion, we introduce the abstraction of a *potential function* to formalize metastable faults. A potential function maps a system's state to a value that quantifies its distance from the good states. A system aiming to converge should take stabilizing actions that drive this potential to zero. Whether every self-stabilizing system can be expressed using potential functions remains an open question for future work.

Definition 1 (Potential function). For a system S with state space Σ and a state predicate G representing a set of good states, a function $f : \Sigma \to \mathbb{R}_{\geq 0}$ is a *potential function* for (S,G) iff:

P1 for all $s \in \Sigma$, $f(s) = 0 \Leftrightarrow s \in G$; and

P2 for all s, $s' \in \Sigma$ and $\alpha \in \mathcal{A}$, if the system makes a transition $s \to^{\alpha} s'$, then $f(s') \leq f(s)$.

In other words, the potential function assigns zero to good states and positive values elsewhere, and every system action can only maintain or reduce this potential. The sole source of destabilization is the environment. If the environment's destabilizing influence exceeds the system's capacity to compensate, stabilization fails. Therefore, correctness for stabilizing systems follows an assume-guarantee form [37]: starting from any state, the system will stabilize *as long as the environment is well-behaved.*

Definition 2 (Stabilizing system). For a system S, a predicate G, and a potential function f for the pair (S,G), the pair (S,f) is *stabilizing* iff there exists an environment predicate E such that, as long as the environment takes actions such that the system state repeatedly satisfies E, eventually the system state also satisfies $f = 0$, and keeps satisfying $f = 0$ as long as environment actions keep maintaining E.

A stabilizing system need not satisfy $f = 0$ initially. It suffices that, if the environment behaves well for long enough, there exist a time after which $f = 0$ holds as long as the environment keeps behaving well. Inspired by the $\to^+$ temporal modality of Abadi and Lamport [12], we use the symbol $\rightsquigarrow^+$ to denote our modality of choice: a pair (S,f) is stabilizing iff there exists an E such that $E \rightsquigarrow^+ f = 0$. We present a formal semantics for $\rightsquigarrow^+$ in Section A.2 of the supplementary materials.

Running Example. Our retry storm implementation (§2.2) naturally decomposes into two systems: the server, $S_1 = (\Sigma_1, \mathcal{A}_1)$, and the retrier, $S_2 = (\Sigma_2, \mathcal{A}_2)$. For S_1, a natural Σ_1 is the state space of the server's queue, *i.e.*, each state is an ordering of requests and retries. We pick one execution of the task `serve` as the only action for the server: $\mathcal{A}_1 = \{\texttt{serve}\}$. This action processes $\min\{s, Q\}$ items per step (s is service capacity, Q the queue size) and sends acknowledgments. The environment injects requests and retries. Since the server repeats `serve` to drain load, a natural potential is $f_1 = \max\{0, Q - s\}$, *i.e.*, the overload on the server. Choosing $f_1 = Q$ would prevent stabilization at zero, because new requests always arrive. f_1 is a potential function, as `serve` never increases it; only the environment's actions do so.

For S_2, a natural Σ_2 consists of pending requests and their timers. It aims to converge to a state where the only client requests pending are the newest. We thus define $f_2 = \max\{0, P - P_{\text{threshold}}\}$, where P is the number of pending requests and $P_{\text{threshold}}$ is a cutoff. [3]

As for the retrier's actions, we split the task `manage` into two parts, one adding requests to `pending` and one removing requests from `pending`. We group the latter and the task `retry` into one action: $\mathcal{A}_2 = \{\texttt{remove-n-retry}\}$. This action never increases f_2: it removes acknowledged requests, updates timers for the ones in `pending`, and retries timed-out ones. Besides submitting acknowledgments, the environment takes care of adding new requests to `pending`, thus increasing f_2.

To show that the server and the retrier are stabilizing, we have to determine environment predicates E_1 and E_2 such that $E_1 \rightsquigarrow^+ f_1 = 0$ and $E_2 \rightsquigarrow^+ f_2 = 0$, *i.e.*, environments under which server and retrier stabilize. If server's environment E_1 always supplies fewer than s requests and retries combined per step, the service rate exceeds the input rate and the server will eventually reach $f_1 = 0$. For the retrier, if E_2 provides more acknowledgments than new client requests per step, the retrier will remove pending requests faster than they accumulate, driving f_2 to zero.

4.2 Compatibility

When composing stabilizing systems, it must be *possible* for them to stabilize together. We capture this sanity check with the notion of *compatibility*: a set of stabilizing systems are compatible if, assuming that each system's writing neighbors have stabilized, the system itself can also stabilize under a well-behaved environment.

Definition 3 (Compatibility). The stabilizing systems (S_1, f_1), (S_2, f_2), ..., and (S_n, f_n) are *compatible* if there exists an environment predicate E such that the following holds for $\|_{0 \leq i < n} S_i$, for all $0 \leq j < n$:

C1 $E \wedge (\bigwedge_{i \in W_j} f_i = 0) \rightsquigarrow^+ f_j = 0$.

We call E a *compatible environment* for the composition.

Compatibility rules out trivially faulty compositions. If two systems cannot assist each other in stabilizing, they are not meant to be composed. We therefore define metastable faults only for compositions of compatible systems.

Running Example. Consider our composition of the server with the retrier: the retrier forwards both client requests and retries to the server, while the server sends acknowledgments back to the retrier. The environment supplies the retrier with client requests; suppose it stops providing

[3] Our analysis shows that $P_{\text{threshold}} = (s - r) \cdot T$ is a suitable pick, with r the nominal client load. Details are omitted for brevity.

new requests. Then the server eventually drains its queue and sends the correponding acknowledgments, leaving the retrier with no pending requests. If we denote this environment E, we have $E \rightsquigarrow^+ f_1 = 0$ and $E \rightsquigarrow^+ f_2 = 0$, which implies $E \wedge f_2 = 0 \rightsquigarrow^+ f_1 = 0$ and $E \wedge f_1 = 0 \rightsquigarrow^+ f_2 = 0$. Since the server's and the retrier's writing neighbors are $W_1 = \{2\}$ and $W_2 = \{1\}$, respectively, the retrier and the server are compatible; they *can* stabilize together in the presence of a well-behaved environment.

4.3 Enter Faults

Compatibility alone does not guarantee joint stabilization. It suffers from a bootstrap problem: following a shock, all systems may start in arbitrary states, with none stabilized. Because stabilization in each system depends on their neighbors effecting a well-behaved environment, the composition may fail to stabilize if no component initially behaves as needed. Indeed, a system S_i might destabilize its reading neighbors while waiting for its writing neighbors to stop destabilizing *it*. When such dependencies form cycles, the system becomes vulnerable to mutual destabilization, even as each component attempts to stabilize. Metastable faults capture this emergent cyclic vulnerability. In order to define a metastable fault, we first need to define a destabilizing action.

Definition 4 (Destabilizing action). Let (S_1, f_1) and (S_2, f_2) be two stabilizing systems in the composition S of some compatible stabilizing systems, where Σ_2 is S_2's state space. Let α_1 be an action with which S_1 writes to S_2, and let $s, s' \in \Sigma_2$. The action α_1 is destabilizing at some state $s \in \Sigma_2$ iff there exists a compatible environment E for S and an E-step $s \rightarrow^{\alpha} s'$ such that either $f_2(s') \geq f_2(s) > 0$ or $f_2(s') > f_2(s) = 0$.

Informally, an action is destabilizing iff it can *possibly* increase or maintain potential somewhere in the system, even in the presence of a compatible environment.

Definition 5 (Metastable fault). Given compatible stabilizing systems $(S_0, f_0), \ldots, (S_{n-1}, f_{n-1})$, their composition $S = \|_{0 \leq i < n} S_i$ *has a metastable fault* iff there exists a cycle of systems in the composition blueprint, such that for all systems in the cycle:

M1 (**Writes-to**) it writes to the next system in the cycle via some action; and

M2 (**Destabilization**) said action is destabilizing at some state of the next system in the cycle.

Thus, the composition of compatible stabilizing systems has a metastable fault iff (*i*) there exists a *writes-to cycle* among them, and (*ii*) each system in the cycle has an action that, when taken in the presence of some compatible environment, either increases the next system's potential when it is zero, or does not decrease it when it is positive.

Locating such faults does not require a global analysis; it replaces reasoning about the entire composition with local, decompositional reasoning about pairs of components. It suffices, for each component, to identify actions that fail to stabilize another given a compatible environment. This involves, for all compatible environments, executing each action from an arbitrary starting state and verifying whether it is destabilizing. A model checker can automate these pairwise checks, after which the designer can determine whether a cycle exists.

Running Example. The retrier–server composition contains a writes-to cycle. The retrier's action `remove-n-retry` never decreases the server's potential f_1, and may even increase it. Similarly, the server's action `serve` can fail to decrease the retrier's potential f_2 when it sends only useless acknowledgments—*e.g.*, when serving retries instead of new requests. Thus, the composition exhibits a metastable fault.

5 Metastable Failures

A metastable failure occurs when a composition of compatible stabilizing systems fails to stabilize under a compatible environment. In other words, some component experiences positive potential infinitely often despite a well-behaved environment and despite its own stabilizing tendency to converge to zero potential. A concise way to express this is via temporal logic [41, 43], where $\Box$ and $\Diamond$ denote the `always` and `eventually` modalities. Combined as $\Box\Diamond P$ for some predicate P, they assert that P happens always eventually, *i.e.*, infinitely often.

Definition 6 (Metastable failure). The composition S of the compatible stabilizing systems (S_0, f_0), (S_1, f_1), ... , and (S_{n-1}, f_{n-1}) has a *metastable failure* iff there exists a compatible environment E such that the composition admits an execution σ satisfying $\Box E \wedge \Box\Diamond\neg(\bigwedge_{0 \leq i < n} f_i = 0)$, *i.e.*, an execution wherein the environment is always compatible but the system experiences positive potential infinitely often. We call σ a *metastable execution* of S.

Metastable faults do not *necessarily* imply metastable failures, *i.e.*, faults are not sufficient for failures. Even as faults show structural destabilizing tendencies between components, compatibility implies the presence also of stabilizing tendencies: whether a potential will remain positive infinitely often depends on the *ordering* of these tendencies during executions. Queues, the OS, and timers are among different parts of a computer system that affect ordering of events; we refer collectively to all such parts of the system that effect such orderings as the *scheduler*.

If the scheduler favors destabilizing actions, potential will stay positive, triggering more destabilizing events and bad scheduling choices in a self-reinforcing loop. A metastable failure will eventually emerge from this cycle of destabilizing actions and poor scheduling. Conversely, if stabilizing actions are prioritized for *long enough*, compatibility will help drive potential to zero. Once one system stabilizes, by compatibility it will in turn help stabilize its reading neighbors, until every component's potential reaches zero. At that point, postponed destabilizing actions cease to exist—as we see in the example

below—as compatibility implies that components do not destabilize each other in stability; all components, therefore, stabilize at zero potential.

Running Example. The pseudo-code in Figure 1 shows the scheduler's role in driving the system toward a metastable failure. The retrier issues retries every T timesteps, while the server places both retries and requests in a single queue and serves them in arrival order. If a shock overloads the server for long enough, the retrier's scheduling policy will cause it to flood the server with retries of pending requests. As retries come to dominate the queue, the server increasingly wastes its resources by serving only old retries for which it has already responded to the original request. The retrier's aggressive scheduler amplifies load; the server's passive scheduler lets this amplification erode effective capacity. That degraded capacity starves the retrier's pending requests, triggering yet another wave of retries and deepening the overload.

A metastable failure can be avoided by changing how components schedule their actions. For example, the server could maintain two queues—one for retries and one for requests—prioritizing requests until enough acknowledgments are sent to the retrier, then switching to retries. This ensures the retrier sees more acknowledgments than new requests, eventually driving the potential to zero; at that point, retries cease to exist as all pending requests have received acknowledgments, and all incoming requests will be served before timing out. Alternatively, the retrier could employ exponential backoff, increasing its timeout until it pauses long enough for the server to clear its backlog and send pending acknowledgments. Either remedy allows the components to act for each other as the stabilizing environment each needs.

5.1 Faults Are Necessary For Failures

Our analysis establishes a fundamental relationship between metastable faults and metastable failures: while metastable faults are not sufficient to induce metastable failures, they are a necessary condition. Specifically, if a composition of compatible stabilizing systems exhibits a metastable failure, then the composition has a metastable fault. Although it may seem intuitive that a metastable failure cannot arise without faults, this is not automatic from our definitions. We state this observation as a theorem together with a proof sketch here, deferring a rigorous proof to Section A.4 of supplementary materials.

Theorem 1. *Let (S_0, f_0), (S_1, f_1), ..., and (S_{n-1}, f_{n-1}) be compatible stabilizing systems, and $S = \|_{0 \le i < n} S_i$ their composition. If S has a metastable failure, then it has a metastable fault.*

Proof Sketch. We prove the contrapositive by induction on n, where we inductively assume that the claim holds for the composition of any $k < n$ compatible stabilizing systems. For the base case $n = 1$, the claim holds trivially. Now for $n > 1$, let E be any compatible environment for the composition S of compatible stabilizing systems (S_0, f_0), (S_1, f_1), ..., and (S_{n-1}, f_{n-1}). If S has no metastable faults, then there should exist some component S_j such that no matter what actions its writing neighbors take in the presence of E, they reduce S_j's potential. Let J be the index set of all such components. Since any joint action of components in W_j is serializable, and S_j does not increase its own potential, we conclude that f_j will eventually reach 0 and stay there, *i.e.*, $E \rightsquigarrow^+ f_j = 0$. This implies $E \rightsquigarrow^+ \bigwedge_{j \in J} f_j = 0$. Now, consider the system with indices not in J: $E \wedge (\bigwedge_{j \in J} f_j = 0)$ is a compatible environment for the remaining composition. We thus have $E \wedge (\bigwedge_{j \in J} f_j = 0) \rightsquigarrow^+ \bigwedge_{j \in [n] \setminus J} f_j = 0$. Putting this besides $E \rightsquigarrow^+ \bigwedge_{j \in J} f_j = 0$, we deduce $E \rightsquigarrow^+ \bigwedge_{0 \le j < n} f_j = 0$, *i.e.*, the composition does not infinitely often experience positive potential in the presence of a compatible environment. □

This result generalizes prior work showing that cycles in the composition blueprint—*i.e.*, feedback loops—are necessary for metastable failures [27]. Our contribution builds on this insight by introducing an automatically verifiable criterion for excluding metastable failures: construct the composition blueprint and check for cycles of destabilizing actions. If no such cycles are present, then, under the assumed potential model, the system is not susceptible to metastable failures.

The proof above relies critically on the assumption that joint actions are serializable. Without this assumption, the potential of a component becomes ill-defined when neighboring components perform concurrent writes. The harmful effect of circular destabilization emerges only when actions have well-defined and isolated impacts on the potential of individual components.

6 Achieving Metastable Fault Tolerance

The previous sections explain metastability using abstractions, yet real failures occur in production systems with tens of thousands of lines of code. Bridging this gap is challenging and demands deep, cross-stack expertise, as some faults couple together components from across the stack. Guided by the goal of analyzing metastability in production code, we take a modest first step by sketching a design methodology for building MFT systems, and apply it to a modeled version of a novel incident (§7). The methodology revolves around proving that a system is MFT: it (*i*) extracts from the system the object of the proof, called the *metastability skeleton*, thereby also eliminating boilerplate code, (*ii*) introduces a way to pick and test candidate potential functions, (*iii*) augments the skeleton with suitable scheduling mechanism and policy, and (*iv*) outlines a proof blueprint. Our methodology draws on experience studying diverse metastable failures, and we have developed the Nyx toolkit[4] to support it.

Step 1: Derive a metastability skeleton. In practice, components destabilize one another by exchanging different types of requests. Upon receiving a request, a component undergoes a sequence of state transitions—possibly altering its potential—and may in turn issue requests to other

[4] Link removed for anonymity.

components. The occurrence of a failure depends on three factors: (*i*) the scheduling of in-flight requests, (*ii*) the scheduling of internal state transitions, and (*iii*) the causal relationships among different request types. An operational model that captures these aspects enables us to isolate what matters for metastability while abstracting away irrelevant system details in the interest of simplicity and tractability. We refer to this model as the *metastability skeleton*.

Our operational model of choice uses queues and resources to implement scheduling; queues schedule requests upon their arrival at the destination, while resources control the scheduling of processes and threads. The skeleton also requires minimal logic for generating and linking different types of requests, *e.g.*, upon receiving a request of type `client`, a server generates a request of type `ack` and sends it back. Queue capacities, resource availability, and timers are especially important in this model, as they govern the flow of requests around the system.

The Nyx DSL (§2.1) exposes these abstractions directly: it supports message passing between agents, provides input and output queues for each agent, enables task scheduling based on resource availability, and offers common programming constructs (*e.g.*, while loops) to implement control logic. By making scheduling of messages and tasks explicit, Nyx pools together parts of a computer system that are usually not present within the same codebase, *e.g.*, it enables the designer to include in the Nyx model some details of OS scheduling that actual production code is oblivious to. Figure 1 in §2.2 is an example Nyx syntax, demonstrating both application logic and task scheduling in an intuitive way.

Step 2: Study dynamics. Metastable failures are, after all, one among many dynamic behaviors of systems; thus, any method that exposes system dynamics should, in principle, reveal metastable failures—even without explicitly labeling them as such. Therefore, a lack of detailed insight into faults does not preclude the ability of visualizing failures at design time, and thus alerting the designer of a potential fault.

One particularly effective method is the use of *vector fields* [36]. A vector field represents the system's tendency at each state—relative to some state function—using a vector. For example, in systems modeled by differential equations, the vector indicates the direction and magnitude of the derivative at each point in the state space. By illustrating dynamic tendencies from arbitrary states, vector fields help analyze behavior after a shock perturbs the system.

It is neither necessary nor feasible to use the full system's state—with queue contents, local variables, and in-flight requests—as the vector field state space. Instead, our vector field represents how the system's *potential* evolves as the system interacts with a compatible environment post-shock. This narrower focus reduces dramatically the size of the space, enabling a 2D or 3D representation of the vector field. Within this reduced space, a metastable failure manifests as two coexisting tendencies: one stabilizing at zero and another diverging toward positive potential. Designers can thus predict metastable failures by inspecting the vector field.

The challenge from a modeling perspective lies in identifying suitable candidates for potential. The potential function for a stabilizing component should derive from the component's specified goal: for example, a server aims to serve load, so excess queue load is a natural candidate; a load balancer seeks to distribute load evenly, so any measure of imbalance is appropriate. Definition 1 further simplifies this task by requiring that a component's actions do not increase its potential; if they do, the component should be divided into smaller units, each with an appropriate potential function.

Similar to Metafor [36], the Nyx toolkit provides a `canvas` mode that enables visualizing a system's vector field—but with an important difference. Metafor derives its vector fields by modeling systems as continuous-time Markov chains, implying that the system has no memory (in the statistical sense). Our experience suggests instead that systems experiencing metastable failures have memory, and in fact that memory plays an important role, because it affects scheduling. Thus, Nyx generates its vector field using the system's executions as its source of ground truth, alleviating the need for explicit mathematical modeling. Designers have to annotate the vector field state variables in the Nyx code expressing their system. Then, Nyx (*i*) executes multiple system runs starting with a random shock, (*ii*) records for each state s the set $\texttt{next}(s)$ of states reached immediately after s across all runs, and (*iii*) computes the system's tendency at s as the average of all states in $\texttt{next}(s)$, represented by a vector from s to this average. Nyx uses lock-step execution semantics to capture dynamic tendencies, mirroring the assumption, implicit in differential equations, that variables update simultaneously based on current values. Additionally, Nyx allows designers to specify the shock using programming constructs, *e.g.*, using the keyword `inject`, the designer can insert extra requests in a server's queue.

Figure 3 illustrates how this visualization can guide designers in identifying the failure for the retry storm example (§2.2); the x-axis represents the retrier's potential (pending requests) and the y-axis represents the server's potential (queue size). Assuming a maximum queue capacity of 500 requests, the system exhibits, as expected, two coexisting yet conflicting tendencies: one toward zero and another diverging toward saturation and ever-increasing pending requests.

Step 3: Manage scheduling. Our characterization of metastability identifies two approaches to achieving metastable fault tolerance: (*i*) eliminate the fault or (*ii*) avoid it through appropriate scheduling. While appealing, the first option is often impractical or even impossible without post-mortem insight. For example, in one incident an air conditioning crash raised the temperature, causing (i) a worker's CPU to overheat, prompting (ii) a thermal manager to put the worker to sleep, triggering (iii) a load balancer to misinterpret the worker as idle and assign to it more load, leading to (iv)

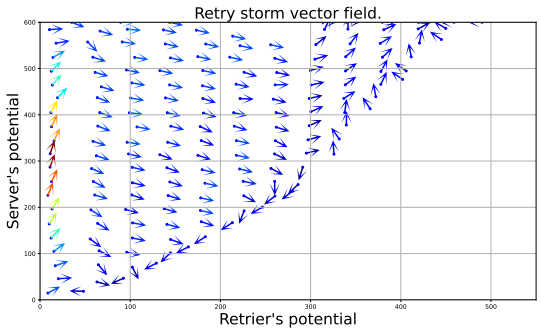


Figure 3. The Nyx vector field for the retry storm. Arrow color indicates tendency intensity at that point; warmer means higher intensity.

sustained overheating due to overload—repeating indefinitely [38]. The fault stemmed from an interaction between load and *temperature*, a factor designers rarely model.

Short of modeling the entire environment, the most viable strategy is to manage scheduling carefully when composing stabilizing systems. This requires explicitly declaring systems as stabilizing where appropriate, which introduces stabilizing and destabilizing interactions. The system's scheduling must satisfy two properties: (**R1**) schedule destabilizing actions tentatively and (**R2**) prioritize stabilizing actions over destabilizing ones until the composition stabilizes. R2 ensures convergence to globally zero potential, while R1—combined with compatibility—guarantees that once the system is globally stable, no further destabilizing actions occur; postponed tentative destabilizing actions disappear. Without this guarantee, scheduling merely reorders increments and decrements, leaving the final potential unchanged. Note that scheduling decisions of import span the entire stack: timers, OS scheduling, network scheduling, etc. The main merit of the metastability skeleton—and Nyx—is striking a balance between generality and feasibility in collecting some of such parts of the system into one specified object of inquiry.

In the retry storm example, exponential backoff is one way to implement R1 and R2: the growing backoff postpones retries until they effectively cease, allowing the server to process pending acknowledgments and clear the retrier's queue, thus altogether eliminating the need for further retries.

Step 4: Complete the proof. A system is metastable-fault-tolerant if it stabilizes despite having a metastable fault. Proving metastable fault tolerance requires showing that, from any arbitrary state that a shock may leave a composition of stabilizing systems in, the system eventually stabilizes globally. The first step is to define the scope of the shock: the most general model would allow a shock to land the system in any arbitrary state, while more realistic models would constrain the state space to those reachable under a specific adversary. The shock has to be finite: there must exist an unknown time after which (*i*) adversarial interactions cease, and (*ii*) all remaining adversarial effects are healed (*e.g.*, crashed nodes are uncrashed). This is akin to the partial synchrony assumption [24] made for consensus protocols, where the network is eventually required to be synchronous forever. Specifying an adversary and proving stabilization establishes, in the same breath, tolerance against all faults triggered by the adversary.

Using the scheduler properties outlined above, the designer must prove that (*i*) the system's potential never increases and (*ii*) sometimes decreases, and that (*iii*) no destabilizing events occur after global stabilization. Together, these guarantees imply metastable fault tolerance; techniques from the self-stabilization literature can be of great help here. In §7, we apply this methodology to design a metastable-fault-tolerant cluster manager for a novel incident.

7 Case Study: Oscillating Membership

We use an incident from a major commercial gaming platform with hundreds of millions of users to show our methodology in action. Consider a cluster of workers $w_0, w_1, \ldots, w_n$ managed by a cluster manager (`CM`). Each worker is in one of three states: active, idle, or crashed. Active workers serve client requests; idle workers do not; crashed workers remain crashed until the environment restores them to idleness. Client load originates from an external load balancer (not modeled here) and is inversely proportional to the number of active workers W: as W decreases, the load per active worker increases.

`CM` aims to maintain at least W_{min} active workers to tolerate crashes. It listens for periodic heartbeats from workers it considers active and, upon a timeout, issues (*i*) a `sleep` command to the unresponsive worker and (*ii*) a `wakeup` command to an idle worker. Workers transition to active upon receiving `wakeup` and to idle upon receiving `sleep`.

During the incident metastability manifested as persistent oscillation in W. After a shock that crashed some workers, the cluster manager failed to restore W to W_{min}; instead, W oscillated as workers were repeatedly put to sleep and awakened. This behavior persisted even after crashed workers recovered, and packet loss and queueing delays did not play a role. Nothing apparent in the worker or cluster manager implementations suggests the root cause.

The Culprits. Two implicit aspects of the system interact to create a cycle leading to oscillation: (*i*) overloaded active workers miss heartbeats (*e.g.*, the service thread starves the heartbeat thread), and (*ii*) `wakeup` commands take longer to execute than `sleep` commands (*e.g.*, waking up involves provisioning virtual machines, whereas sleeping is a simple state change). The failure unfolds as follows: after some workers crash, (*i*) remaining active workers become overloaded; (*ii*) `CM` detects crashes and issues `wakeup` commands; (*iii*) before new workers activate, overloaded workers miss heartbeats, causing `CM` to put them to sleep; (*iv*) new workers finally

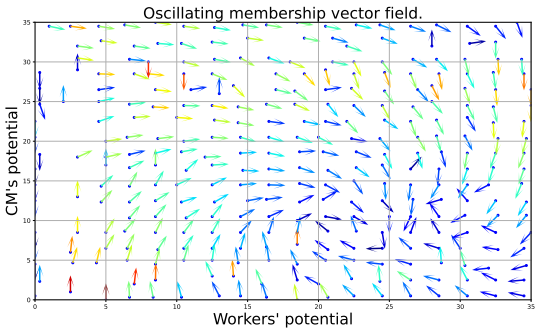


Figure 4. The Nyx vector field for the oscillating membership case.

wake up, only to become overloaded because the old ones are now idle—this patterns repeats indefinitely.

Potential Functions. Both workers and CM respond to a shock as stabilizing systems: workers aim to handle excess load, and CM seeks to maintain sufficient active workers. While load is a natural candidate for worker potential, a useful proxy is $f_1 = W_{noOverload} - W$, where $W_{noOverload}$ is the minimum number of active workers required to avoid missed heartbeats. Workers require an environment that activates enough peers to drive f_1 to zero—here, CM and the crashing environment together form that environment. For CM, let W' denote the number of workers it *deems* active; then $f_2 = W_{min} - W'$ is a suitable potential.[5] Notably, CM cannot reduce f_2 by its own actions, and relies on timely heartbeats to stabilize. The functions f_1 and f_2 are immediately useful for checking compatibility and exposing the fault.

Compatibility. Let E denote an environment that never crashes workers. Under E, if $f_2 = 0$ holds, then CM believes enough workers are active, implying it receives sufficient heartbeats and thus issues no `sleep` commands. Consequently, $f_1 = 0$; formally, $E \wedge f_2 = 0 \rightsquigarrow^+ f_1 = 0$. Similarly, if $f_1 = 0$ holds under E, workers are never overloaded, and if $f_2 > 0$, CM will activate additional workers until $f_2 = 0$; thus, $E \wedge f_1 = 0 \rightsquigarrow^+ f_2 = 0$. This establishes compatibility.

Metastable Fault. Workers destabilize CM by missing heartbeats, while CM destabilizes workers by issuing `sleep` commands. Model CM's actions as `wakeup`(i) and `sleep`(i): the former, if w_i is idle, decrements f_1 (a stabilizing action), while the latter, if w_i is active, increments f_1 (a destabilizing action). A worker action `miss`(i) increments f_2 by making CM believe i has crashed. The possibility of a worker missing heartbeats—and thus destabilizing CM—is inherent in the functional composition of workers and CM; indeed, CM anticipates the possibility of missed heartbeats due to crashes. Locating the fault therefore requires only a careful static analysis of the interface between the components.

[5] No action of CM increases f_2, as `sleep` commands decrease W and not W'.

Metastable Failure. The schedulers turn the fault into a failure: overloaded workers starve heartbeats, and CM, by issuing `sleep(i)` and `wakeup(i)` simultaneously, implicitly allows `sleep` to take effect before `wakeup`. This time gap gives the scheduler of the remaining active workers, which are now overloaded, time to destabilize CM by in turn missing heartbeats, prompting further `sleep` commands—still within the time gap between `sleep` and `wakeup` commands. By the time the first `wakeup` occurs, the system has drifted even farther from stability.

7.1 An MFT Cluster Manager

We apply the method from §6 to this incident.

Step 1: Derive metastability skeleton. We implement a system similar to the one we received the report on using the Nyx DSL. The implementation has two agents, one for CM (129 LoC) and one representing the entire ensemble of workers (116 LoC). The workers' agent simulates the wakeup delay, and also omits heartbeats if $W < W_{noOverload}$. Bundling workers together makes it easier to detect overload by just counting their number and abstracting load away. Designers can also choose to implement each worker as a separate agent, and to use internal scheduling to starve heartbeats; the results would not change. The idiomatic way of writing Nyx code helps the designer specify the metastability skeleton.

Step 2: Study dynamics. Figure 4 shows the resulting vector field for a cluster of 40 workers with fixed timeouts. Nyx generated this field from 1,000 executions, each simulating 100 seconds, in about three minutes. The field reveals two opposing tendencies, one pulling toward zero potential and the other pushing away, resulting in a persistent oscillation. This visualization makes the risk of oscillation immediately apparent and prompts deeper reasoning about the scheduler.

Step 3: Manage scheduling. We demonstrate that it is possible to build an MFT cluster manager for this incident *without* knowing that workers starve heartbeats, *i.e.*, by focusing on the cluster manager as a stabilizing system. We make a simplifying assumption: overload is an instantaneous function of W—the moment $W_{min} - W = 0$, active workers are no longer overloaded.

The cluster manager (CM) issues two commands: `sleep`, which is destabilizing, and `wakeup`, which is stabilizing. Our methodology (§6) rests on two principles: destabilizing actions must be tentative (**R1**), and stabilizing actions must take precedence (**R2**). To enforce these principles, we introduce $\text{CM}_{\texttt{mft}}$, a cluster manager that, by an appropriate combination of new mechanisms and policies, achieves metastable fault tolerance even when workers starve heartbeats. Importantly, $\text{CM}_{\texttt{mft}}$'s new scheduling policies and mechanisms do not alter the cluster manager's functionality: they only control the timing of actions.

New scheduling mechanisms and policy Because $\text{CM}_{\texttt{mft}}$ cannot control how the environment orders `sleep` and `wakeup`

events, the designer must estimate empirically the time needed by each command to take effect, denoted, respectively, as $d_{\texttt{sleep}}$ and $d_{\texttt{wakeup}}$. For simplicity, we assume $d_{\texttt{sleep}} = 0$.

To satisfy R1 and R2, $\texttt{CM}_{\texttt{mft}}$'s scheduling mechanisms ensure that `sleep` events remain tentative, that they can be delayed until sufficient `wakeup` events occur, and that all `wakeup` events target idle workers. Policy complements this by guaranteeing that delays for `sleep` events are long enough that they will follow `wakeup` events. To achieve this, $\texttt{CM}_{\texttt{mft}}$ introduces an artificial delay $d'_{\texttt{sleep}}$; without it, `sleep` would precede `wakeup` since $d_{\texttt{wakeup}} > d_{\texttt{sleep}} = 0$.

A Refined View of Workers' States To implement these guarantees, $\texttt{CM}_{\texttt{mft}}$ refines its view of worker states. Instead of modeling them as either `idle` or `active`, it uses five states: {`idle`, `active`, `waking-up`, `snoozing`, `pending-timeout`}. When a worker w_i times out, $\texttt{CM}_{\texttt{mft}}$ checks for an idle worker w_j. If one exists, it issues `wakeup`(j) and marks w_j as `waking-up`. Upon confirmation, w_j becomes `active`; otherwise, it reverts to `idle`. Simultaneously, w_i moves to `snoozing`, but `sleep`(i) is delayed by $d'_{\texttt{sleep}}$. If no idle worker exists, w_i enters `pending-timeout`, and $\texttt{CM}_{\texttt{mft}}$ retries until a heartbeat arrives or an idle worker is found.

The intermediate states `waking-up` and `snoozing` are essential to prevent issuing `wakeup` to non-idle workers. If workers were modeled only as either `idle` or `active`, overlapping `sleep`/`wakeup` actions could cause a `wakeup` command to be sent to an overloaded worker, hence resulting in no decrease in potential. Similarly, `pending-timeout` ensures eventual recovery to $W_{\min}$ active workers by enabling $\texttt{CM}_{\texttt{mft}}$ to remember timed out workers and to retry until either idle workers are found or heartbeats resume.

Choosing a delay policy Avoiding metastable failures requires *all* `sleep` events to happen after any `wakeup` event until stability is restored post-shock. Thus, the value of $d'_{\texttt{sleep}}$ is critical. Setting $d_{\texttt{wakeup}} < d'_{\texttt{sleep}}$ may seem sufficient, but it does not guarantee stabilization. It only ensures the correct ordering for every pair of `wakeup` and `sleep` events issued in response to a timeout, and a poor choice of T can still lead to a `sleep` event happening before an unrelated `wakeup` event, and cause a metastable failure. Instead, as we show in the following proof, we require $d_{wakeup} + T < d'_{sleep}$.

The Protocol Protocol 1 illustrates $\texttt{CM}_{\texttt{mft}}$. The protocol proceeds in steps; it comprises three procedures driven by Main, which is executed in each step. Uncrashed runs asynchronously when a worker notifies $\texttt{CM}_{\texttt{mft}}$ that it is idle after a crash; heartbeat handling is omitted for brevity. Two practical observations follow. First, communication between workers and $\texttt{CM}_{\texttt{mft}}$ occurs only during state changes. Since active worker counts oscillated in the reported incident, we infer that this messaging infrastructure is unaffected by overload and thus reliable. Second, although our implementation uses a single thread for timer updates, timers can run concurrently in practice. $\texttt{CM}_{\texttt{mft}}$ itself is not under load and can schedule timers successfully.

Protocol 1 The MFT cluster manager $\texttt{CM}_{\texttt{mft}}$.

```
    // State variables:
 1: w ← [active,...,active,idle,...,idle]          // Initial view of the workers.
 2: w' ← [idle,...,idle]                           // Next view of the workers.
 3: timers ← [0,0,...,0]                           // Initial values of the timers.
 4: ∀i ∈ {idle,active,...} : X_i ← {j | w[j] = i}  // Partitions of worker states.
 5: function GetPartition(i) return {j | w[j] = i}
 6: procedure Uncrashed(j)
 7:     if w[j] ∈ {active,waking-up,pending-timeout} then
 8:         Send(j,wakeup);
 9:         w[j] ← waking-up; timers[j] ← 0;
10:     else
11:         w[j] ← idle; timers[j] ← 0;
12: procedure Timeout(j)
13:     if w[j] ∈ {active,pending-timeout} then
14:         if ∃i : w[i] == idle then
15:             Send(i,wakeup);
16:             w'[i] ← waking-up; timers[i] ← 0; timers[j] ← 0;
17:             return snoozing
18:         else timers[j] = 0; return pending-timeout
19:     else if w[j] == waking-up then timers[j] = 0; return idle
20: procedure Main                                 // Execution starts from here
21:     for i in {idle,active,...} do              // Update the partitions.
22:         X_i ← getPartition(i);
23:     timers ← updateTimers(timers);
24:     for j in X_active do
25:         if timers[j] == T then
26:             w'[j] ← Timeout(j);
27:     for j in X_waking-up do
28:         if timers[j] == d_wakeup then
29:             w'[j] ← Timeout(j);
30:     for j in X_snoozing do
31:         if timers[j] == d_sleep then Send(j,sleep);
32:             timers[j] ← 0;
33:     for j in X_pending-timeout do
34:         if timers[j] == T then
35:             w'[j] ← Timeout(j);
36:     w ← w';                                    // Update the worker states
```

Step 4: Complete the proof. For ease of exposition, we make an assumption: $W_{noOverload} = W_{\min}$.

The adversary for $\texttt{CM}_{\texttt{mft}}$ crashes workers during a temporary shock; consequently, all crashed workers must eventually be no longer crashed, though the duration of the shock is unknown. After the shock, the system begins in an arbitrary state that consists of (*i*) workers (that are either idle or active) and (*ii*) $\texttt{CM}_{\texttt{mft}}$'s view of the state of these workers, with the corresponding timers.

Following the methodology in §6, we must show that, from any such initial state: (*i*) $\texttt{CM}_{\texttt{mft}}$ postpones `sleep` commands while the system is stabilizing; (*ii*) $\texttt{CM}_{\texttt{mft}}$ issues enough `wakeup` commands to restore $W = W_{min}$; and (*iii*) once $W = W_{min}$, $\texttt{CM}_{\texttt{mft}}$ ceases issuing `sleep` commands. We prove them in reverse order.

(*iii*): Because overload is an instantaneous function of W, the moment $W = W_{min}$, all active workers are no longer overloaded and respond to $\texttt{CM}_{\texttt{mft}}$ with heartbeats. At that point, $\texttt{CM}_{\texttt{mft}}$ knows that at least W_{min} workers are active and will never issue `sleep` events again.

(*ii*): Let X_i denote the set of workers that $\texttt{CM}_{\texttt{mft}}$ deems in state i, for $i \in$ {`idle`, `active`, `waking-up`, `snoozing`,

pending-timeout}. Define: $Z = |X_{\texttt{active}}| + |X_{\texttt{snoozing}}| + |X_{\texttt{pending-timeout}}|$, $V = |X_{\texttt{active}}| + |X_{\texttt{waking-up}}| + |X_{\texttt{snoozing}}| + |X_{\texttt{pending-timeout}}|$, and $U = |X_{\texttt{active}}| + |X_{\texttt{waking-up}}| + |X_{\texttt{pending-timeout}}|$. After the shock, workers notify $\texttt{CM}_{\texttt{mft}}$ upon recovering and are then placed in state waking-up. If $\texttt{CM}_{\texttt{mft}}$ deems a worker w_i in X_j for $j \in$ {active, snoozing, pending-timeout}, then w_i's actual state is active. Thus, after the shock, $Z = W$ holds invariantly. Furthermore, executions satisfy the invariant $U = W_{min}$. The invariant holds initially, because the system starts with W_{min} active workers and the rest idle. Subsequently, workers in $X_{\texttt{pending-timeout}}$ either remain or move to $X_{\texttt{waking-up}}$; workers in $X_{\texttt{waking-up}}$ either remain or move to $X_{\texttt{active}}$; and whenever a worker leaves $X_{\texttt{active}}$, it either moves to $X_{\texttt{pending-timeout}}$ or to $X_{\texttt{snoozing}}$ while another worker moves to $X_{\texttt{waking-up}}$. Therefore, U never changes, so $U = W_{min}$ throughout. Since $V \geq U = W_{min}$, we have $|X_{\texttt{waking-up}}| \geq W_{min} - Z$. Because $Z = W$, it follows that $|X_{\texttt{waking-up}}| \geq W_{min} - W$: there always are enough workers scheduled to wake up.

(*i*): Note that at all times the difference between the wake-up times of any two workers in $X_{\texttt{waking-up}}$ is at most T. This holds because $\texttt{CM}_{\texttt{mft}}$ checks heartbeats every T timesteps and retries workers in pending-timeout every T timesteps. Since $d_{\texttt{wakeup}} + T < d_{\texttt{sleep}}$, whenever a worker enters snoozing, its corresponding sleep event is scheduled only after all pending wakeup events have occurred.

We apply our methodology to the look-aside cache case study [17, 35] as well, and defer it to Section B of supplementary materials for lack of space.

8 Related Work

Metastable failures were first introduced by Bronson et al. [17], who informally characterize them as persistent, self-sustaining degraded modes of operation. Huang et al. [35] expand on this work by presenting a broader suite of industrial metastability incidents, defining metastability in terms of persistent overload after a trigger, and categorizing self-sustaining mechanisms as either workload amplification or capacity degradation. Habibi et al. [33] focus on the retry storm incident, and leverage queuing theory and continuous-time Markov chains to explain how repeated request retries destabilize the system. Isaacs et al. [36] reiterate metastable failures as self-sustaining congestive collapse, and present a host of tools ranging from a discrete-event simulator to production-level software to study the effects of various system parameters on emergent self-sustaining overload. Farahbakhsh et al. [27] provide an operational system model involving queues, and define metastability as never-ending difference between the outputs of two copies of a system: one starting after a shock, and one starting from the normal initial state. Anand [14] propose a toolchain to develop microservice-based systems in a modular, plug and play, and configurable fashion, and use their framework to demonstrate metastable failures in the deathstarbench suite [31]. While insightful, these works (*i*) do not characterize the underlying causes of metastable failures, and some (*ii*) cannot account for important metastability incidents that do not manifest as overload.

We have identified metastability incidents that prior studies have overlooked. Floyd and Van Jacobson [29] model router synchronization using Markov chains, showing how message exchange can synchronize and overload the network. Khan et al. [38] describe a vicious cycle between a load balancer and a thermal manager, triggered by an AC failure. Qian et al. [44] report a Hadoop [2] incident where a missed heartbeat causes recovery actions to interfere with heartbeat management, exposing harmful interactions between error handling and request handling. Ford [30] documents oscillations caused by faulty coupling between a load balancer and a power optimizer.

Distributed systems exhibit many failure modes, *e.g.*, crash [28], fail-stop [45], fail-slow [22, 32], and Byzantine faults [39]. Metastability, however, is fundamentally different: it does not stem from permanent component failures but is an *emergent property* sustained by feedback loops spanning individually correct components.

Classic examples of failures caused by component interactions rather than local faults include deadlocks, livelocks, and race conditions. Deadlocks and livelocks arise from cyclic dependencies among processes [46]. Deadlocks can be characterized using resource-allocation graphs [19] and avoided via algorithms such as Banker's algorithm [21] or prevented entirely using lock-ordering disciplines [34]. Concurrent accesses to shared resources can lead to nondeterministic failures [25] and require atomicity. Metastable faults also stem from cyclic dependencies, expressed through the concept of stabilization. The solution, as in deadlock, lies in scheduling.

Self-stabilizing protocols were first introduced by Dijkstra [20]. Self-stabilizing systems converge to a set of good states and remain there. Metastable faults and failures arise when such systems are composed.

Recent work has explored correct-by-design cluster management controllers. Sun et al. [47] propose eventually stable reconciliation (ESR), requiring controllers to eventually drive the system to the desired state and keep it there. They present Anvil, a tool for designing controllers and proving correctness using Verus [40]. While ESR resembles stabilization, their framework excludes metastability: it assumes components stop destabilizing the controller until it updates the state, whereas metastability occurs precisely when such problematic interactions persist.

9 The Road Ahead

We introduce the first analytical characterization of metastable faults and failures, and present a methodology for designing MFT systems. We apply our methodology to three case studies, showing the generality of our approach.

Our contributions set the stage for future work on building a toolbox to prevent, detect, and mitigate metastable failures.

References

[1] Google compute engine incident #19008. https://status.cloud.google.com/incident/compute/19008.

[2] HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[3] Incident management at spotify. https://engineering.atspotify.com/2013/6/incident-management-at-spotify.

[4] Memcached. https://memcached.org/.

[5] MySQL. https://www.mysql.com/.

[6] NGINX. https://nginx.org/.

[7] PHP. https://www.php.net/.

[8] Retry storm antipattern. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/retry-storm/.

[9] Summary of the amazon dynamodb service disruption and related impacts in the us-east region. https://aws.amazon.com/message/5467D2/.

[10] Summary of the amazon ec2 and amazon rds service disruption in the us east region. https://aws.amazon.com/message/65648/.

[11] Summary of the aws service event in the northern virginia (us-east-1) region. https://aws.amazon.com/message/12721/.

[12] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–535, 1995.

[13] B. Alpern and F. B. Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[14] V. Anand, D. Garg, A. Kaufmann, and J. Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 482–497, 2023.

[15] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[16] N. Bronson. Solving the mystery of link imbalance: A metastable failure state at scale. https://engineering.fb.com/2014/11/14/production-engineering/solving-the-mystery-of-link-imbalance-\a-metastable-failure-state-at-scale/.

[17] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu. Metastable failures in distributed systems. In *Proc. HotOS*, 2021.

[18] M. Brooker. The hardest problem (and the silliest). https://brooker.co.za/blog/resources/mbrooker_socc23_stability.pdf.

[19] E. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM CSUR*, 3(2), 1971.

[20] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, Nov. 1974.

[21] E. W. Dijkstra. *The mathematics behind the banker's algorithm*, chapter EWD623. Springer-Verlag, Berlin, Heidelberg, 1982.

[22] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[23] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[24] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[25] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 237–252, New York, NY, USA, 2003. Association for Computing Machinery.

[26] S. Estyak. Metastability. https://github.com/SalmanEstyak/Metastability/tree/main, 2022.

[27] A. Farahbakhsh, A. Haeberlen, Q. Lu, L. Alvisi, R. Van Renesse, and S. Cohen. Modeling metastability. In *2025 Workshop on Hot Topics in Networks*, 2025.

[28] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[29] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. In *Conference proceedings on Communications architectures, protocols and applications*, pages 33–44, 1993.

[30] B. Ford. Icebergs in the clouds: the other risks of cloud computing. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.

[31] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[32] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[33] F. Habibi, T. Lorido-Botran, A. Showail, D. C. Sturman, and F. Nawab. MSF-Model: Modeling metastable failures in replicated storage systems. *CoRR*, abs/2309.16181, 2023.

[34] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[35] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *Proc. OSDI*, 2022.

[36] R. Isaacs, P. Alvaro, R. Majumdar, K. Kumar, M. Reddy, M. Salamati, and S. Soudjani. Analyzing metastable failures. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 172–178, 2025.

[37] C. B. Jones. Specification and design of (parallel) programs. In *9th IFIP World Computer Congress (Information Processing 83)*. Newcastle University, 1983.

[38] M. M. H. Khan, J. Heo, S. Li, and T. Abdelzaher. Understanding vicious cycles in server clusters. In *2011 31st International Conference on Distributed Computing Systems*, pages 645–654. IEEE, 2011.

[39] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. In *Concurrency: the works of Leslie Lamport*, pages 203–226. ACM Books, 2019.

[40] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.

[41] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems: Safety, 1995.

[42] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[43] A. Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. ieee, 1977.

[44] S. Qian, W. Fan, L. Tan, and Y. Zhang. Vicious cycles in distributed software systems. In *Proc. 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2014.

[45] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.

[46] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018.

[47] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, et al. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, 2024.

[48] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, volume 8, pages 281–294, 2008.

A Model and Formal Semantics

This section includes our formal model, the semantics for the $\rightsquigarrow^+$ operator, our definitions in the paper for ease of access, and the proof for Theorem 1 from the paper.

A.1 Model

To reason about how components destabilize each other, we need a model that captures state changes, action effects, and how components read and write shared state.

A system $S = (\Sigma_S, \mathcal{A}_S, \mathcal{V}, \mathcal{V}_E)$ is a state machine with a state space Σ_S, a set of actions $\mathcal{A}_S$, a set of variables $\mathcal{V}$, and a set of variables $\mathcal{V}_E$ such that $\mathcal{V}_E \subseteq \mathcal{V}$—these are variables that the environment in which the system operates can modify. Each state is a valuation of the variables in $\mathcal{V}$. S interacts with an environment that can modify the variables in $\mathcal{V}_E$ by taking environment actions. Environment actions are arbitrary—the system does not control the environment's behavior. We denote state transitions of the system as $s \rightarrow s'$ of the system, where $s,s' \in \Sigma_S$. We omit the subscript S when the system is clear from the context.

For a system $S = (\Sigma, \mathcal{A}, \mathcal{V}, \mathcal{V}_E)$, a predicate P is a subset of Σ; an environment predicate Q is a predicate that involves only variables in $\mathcal{V}_E$. Any execution σ of the system is a trace $\sigma = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$, where $s_i \in \Sigma$ for all $i \geq 0$ and in each transition at least one of system and environment takes an action. For every execution σ of the system, all suffixes of σ are also executions of the system. Given some environment predicate E and system states s and s', a transition $s \rightarrow s'$ is an E-step if s and s' satisfy E. If during a transition $s \rightarrow s'$ only the system takes an action, say α, we further label the transition as $s \rightarrow^{\alpha} s'$. To keep notation short, we will omit mention of variables and write $S = (\Sigma, \mathcal{A})$; it is to be inferred that the context implicitly indicates which variables the environment can modify.

If a system assigns a value to a variable following an action, we say the system *writes to* the variable via the action, establishing a writes-to relation between them; if a system reads the value of a state variable with an action, we say that the system *reads from* the variable via the action.

Let $\perp$ denote the absence of an action for any system. Given two systems $S_1 = (\Sigma_1, \mathcal{A}_1, \mathcal{V}_1, \mathcal{V}_E^1)$ and $S_2 = (\Sigma_2, \mathcal{A}_2, \mathcal{V}_2, \mathcal{V}_E^2)$, we define their composition as the system $S_1 \|_{\mathcal{V}'} S_2 = (\Sigma_1 \times \Sigma_2, \mathcal{A}, \mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{V}')$, where $\times$ denotes the Cartesian product and $\mathcal{V}' \subseteq \mathcal{V}_E^1 \cup \mathcal{V}_E^2$. As for $\mathcal{A}$, assuming $\alpha \in \mathcal{A}_1$ and $\beta \in \mathcal{A}_2$, each action $\gamma \in \mathcal{A}$ is of one of (i) (α,β), (ii) $(\alpha,\perp)$, and (iii) $(\perp,\beta)$: the composition makes a state transition whenever at least one component takes an action. The set $\mathcal{V}'$ specifies the remaining environment variables in the composition, since components can partially be the environment for each other. If, upon composition, S_1 becomes responsible for writing to some variable of S_2 that was previously written to by the environment, the environment stops writing to that variable in the composition. To keep notation short, we will drop the subscript $\mathcal{V}'$; the specifics of which component writes to which environment variable of another component are to be inferred from the context. We generalize this to compositions of more than two systems, and denote the composition of the n systems $\{S_i\}_{0 \leq i < n}$ with $\|_{0 \leq i < n} S_i$. Whenever the composition S takes an action $\alpha = (\alpha_0, \ldots, \alpha_{n-1})$ and the environment takes an action e, the transition is serializable [15, 42], *i.e.*, the resulting state is equivalent to that resulting after *some* serial execution of the actions $\{\alpha_i\}_{0 \leq i < n}$ and e.

Given a composition of n systems $\{S_i\}_{0 \leq i < n}$, we lift the writes-to relation between systems and variables to a writes-to relation between systems. If a system S_i, via some action $\alpha_i \in \mathcal{A}_i$, writes to a state variable of S_i that system S_j reads from, or directly to a state variable of S_j that S_j reads from, we say that S_i writes to S_j via α_i. This relation induces a *composition blueprint*: a directed graph whose vertices are systems and whose edges represent writes-to relations between them. If S_i writes to S_j, the edge is from S_i to S_j. For a system S_i, we call the set of all systems that write to it its *writing neighbors*, and denote it with W_i. Similarly, we call the set of all systems that S_i writes to as its *reading neighbors*, and denote it with R_i.

To facilitate our proofs, which entail liveness assertions, we assume that in every execution of a composition S (i) each component takes actions infinitely often, and (ii) during each transition at least one writing neighbor of every component takes an action.

A.2 Semantics for $\rightsquigarrow^+$

Let $S = (\Sigma, \mathcal{A})$ be a system interacting with some environment, $E \subseteq \Sigma$ be an environment predicate, and $G \subseteq \Sigma$ a predicate. A state $s \in \Sigma$ satisfies a predicate P, *i.e.*, $s \models P$, if $s \in P$. The temporal formula $E \rightsquigarrow^+ G$ is an assertion on the executions of S as it is interacting with its environment. Consider the execution $\sigma = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$, where $s_i \in \Sigma$ for all $i \geq 0$. We define $\sigma_{i:\infty}$ to be the execution $s_i \rightarrow s_{i+1} \rightarrow \ldots$. σ satisfies $E \rightsquigarrow^+ G$, *i.e.*, $\sigma \models E \rightsquigarrow^+ G$, if the following holds:

$$\exists\, i : [\bigwedge_{0 \leq j \leq i} (s_j \models E)] \Rightarrow [(s_i \models G) \wedge (\sigma_{i+1:\infty} \models G\, \mathcal{U}\, \neg E)],$$

where $\mathcal{U}$ denotes the `unless` temporal modality [41]. The system S satisfies $E \rightsquigarrow^+ G$, *i.e.*, $S \models E \rightsquigarrow^+ G$, if all of its executions satisfy $E \rightsquigarrow^+ G$.

We use the following property of the $\rightsquigarrow^+$ operator in the proof of Theorem 1 (§A.4).

Lemma 1. *Let $S = (\Sigma, \mathcal{A})$ be a system, and P, Q, and R predicates over Σ. If $S \models P \wedge Q \rightsquigarrow^+ R$ and $S \models P \rightsquigarrow^+ Q$, then $S \models P \rightsquigarrow^+ Q \wedge R$.*

Proof. Consider some execution $\sigma = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$ of S. Since $S \models P \rightsquigarrow^+ Q$, there exists some i such that, if all $s_j \models P$ for $0 \leq j \leq i$, then $s_i \models Q$ and $\sigma_{i+1:\infty} \models Q\, \mathcal{U}\, \neg P$. That is, after index i, Q keeps holding in every state of σ unless P stops holding at some point. Similarly, since $S \models P \wedge Q \rightsquigarrow^+ R$, therefore $\sigma_{i+1:\infty} \models P \wedge Q \rightsquigarrow^+ R$—note that $\sigma_{i+1:\infty}$ is a suffix of σ, and is therefore an execution of S. Thus, there exists some k such

that, if all $s_l \models P \wedge Q$ for $i+1 \le l \le k$, then $s_k \models R$ and $\sigma_{k+1:\infty} \models R \,\mathcal{U}\, \neg(P \wedge Q)$. Based on the above, if for all $0 \le n \le k$ we have $\sigma_n \models P$, it holds that $\sigma_k \models Q$ and $\sigma_{k+1:\infty} \models Q \,\mathcal{U}\, \neg P$. Similarly, we infer that $\sigma_k \models R$ and $\sigma_{k+1:\infty} \models R \,\mathcal{U}\, \neg(P \wedge Q)$. Based on the properties of the `unless` operator, we infer that $\sigma_{k+1:\infty} \models (Q \wedge R) \,\mathcal{U}\, \neg P$, and we independently establish $\sigma_k \models Q \wedge R$. This proves that $\sigma \models P \rightsquigarrow^+ Q \wedge R$. Since we picked σ arbitrarily, we thus have $S \models P \rightsquigarrow^+ Q \wedge R$. □

A.3 Faults and Failures

We provide our definitions for ease of accessibility when reading the proof.

Definition 7 (Potential function). For a system $S = (\Sigma, \mathcal{A})$ and a state predicate $G \subseteq \Sigma$ representing a set of good states, a function $f : \Sigma \to \mathbb{R}_{\ge 0}$ is a *potential function* for (S,G) iff:

P1 for all $s \in \Sigma$, $f(s) = 0 \Leftrightarrow s \in G$; and

P2 for all $s, s' \in \Sigma$ and $\alpha \in \mathcal{A}$, if the system makes a transition $s \to^{\alpha} s'$, then $f(s') \le f(s)$.

Definition 8 (Stabilizing system). For a system $S = (\Sigma, \mathcal{A})$, a predicate $G \subseteq \Sigma$, and a potential function $f : \Sigma \to \mathbb{R}_{\ge 0}$ for the pair (S,G), the pair (S,f) is *stabilizing* iff there exists an environment predicate E such that, as long as the environment takes actions such that the system state repeatedly satisfies E, eventually the system state also satisfies $f = 0$, and keeps satisfying $f = 0$ as long as environment actions keep maintaining E.

Definition 9 (Compatibility). The stabilizing systems (S_1, f_1), (S_2, f_2), ..., and (S_n, f_n) are *compatible* if there exists an environment predicate E such that the following holds for $\|_{0 \le i < n} S_i$, for all $0 \le j < n$:

C1 $E \wedge (\bigwedge_{i \in W_j} f_i = 0) \rightsquigarrow^+ f_j = 0$.

We call E a *compatible environment* for the composition.

Definition 10 (Destabilizing action). Let (S_1, f_1) and (S_2, f_2) be two stabilizing systems in the composition S of some compatible stabilizing systems, where $S_i = (\Sigma_i, \mathcal{A}_i)$ for $i \in \{1,2\}$. Let $\alpha_1 \in \mathcal{A}_1$ be an action with which S_1 writes to S_2. If there exists a compatible environment E for S and states $s, s' \in \Sigma_2$ such that (i) $s \to s'$ is an E-step, and (ii) either $f_2(s') \ge f_2(s) > 0$ or $f_2(s') > f_2(s) = 0$, then we say that α_1 is destabilizing at s.

Definition 11 (Metastable fault). Let $i \oplus 1$ denote $i + 1 \bmod k$ for $0 \le i \le k-1$ and some k. Given compatible stabilizing systems (S_0, f_0), ..., (S_{n-1}, f_{n-1}), where $S_i = (\Sigma_i, \mathcal{A}_i)$, their composition $S = \|_{0 \le i < n} S_i$ *has a metastable fault* iff there exists a cycle $M = \{S_{i_0}, \ldots, S_{i_{k-1}}\}$ of systems in the composition blueprint, and an action $\alpha_{i_j} \in \mathcal{A}_{i_j}$ for $0 \le j \le k-1$, such that for all $0 \le j < k$:

M1 (**Writes-to**) each S_{i_j} writes to $S_{i_{j\oplus 1}}$ via α_{i_j}; and

M2 (**Destabilization**) there exists a state $s_{i_{j\oplus 1}} \in \Sigma_{i_{j\oplus 1}}$, such that $f_{i_{j\oplus 1}}(s_{i_{j\oplus 1}}) > 0$ and α_{i_j} is destabilizing at $s_{i_{j\oplus 1}}$.

A.4 Proof of Theorem 1

We begin by proving two lemmas.

Lemma 2. *Let (S_0, f_0), (S_1, f_1), ... , and (S_{n-1}, f_{n-1}) be compatible stabilizing systems, S be their composition, and J be a subset of $\{0, \ldots, n-1\}$. If E is a compatible environment for S, then $E \wedge (\bigwedge_{j \in J} f_j = 0)$ is a compatible environment for the systems with indices in $[n] \setminus J$.*

Proof. Note that, for all i:

$$E \wedge \Big(\bigwedge_{k \in W_i} f_k = 0\Big) \rightsquigarrow^+ f_i = 0 \equiv$$
$$\Big(E \wedge \Big(\bigwedge_{k \in W_i \cap J} f_k = 0\Big)\Big) \wedge \Big(\bigwedge_{k \in W_i \setminus J} f_k = 0\Big) \rightsquigarrow^+ f_i = 0.$$

Moreover, note that:

$$E \wedge \Big(\bigwedge_{k \in J} f_k = 0\Big) \Rightarrow E \wedge \Big(\bigwedge_{k \in W_i \cap J} f_k = 0\Big).$$

If we consider the systems with indices in J as part of the environment for the remaining systems, then the set of writing neighbors of any remaining component S_i changes from W_i to $W_i \setminus J$. We conclude that $E \wedge (\bigwedge_{j \in J} f_j = 0)$ is a compatible environment for the systems with indices not in J. □

Lemma 3. *Let (S_0, f_0), (S_1, f_1), ..., and (S_{n-1}, f_{n-1}) be compatible stabilizing systems, where $S_i = (\Sigma_i, \mathcal{A}_i)$ for $0 \le i < n$, and E be a compatible environment for $S = \|_{0 \le i < n} S_i$. Consider a transition $s \to s'$ of S wherein the environment takes action e and the writing neighbors of some component S_i take actions $\{\alpha_j\}_{j \in W_i}$, where $\alpha_j \in \mathcal{A}_j$, and assume that the serialization order is $s = s_0 \to^{\beta_1} s_1 \to \cdots \to s_k \to^{\beta_{k+1}} s_{k+1} \to \cdots \to^{\beta_m} s_m = s'$, where $\beta_{k+1} = e$ and the rest of $\{\beta_l\}$ are a permutation of the actions $\{\alpha_j\}_{j \in W_i}$. If $s \to s'$ is an E-step, then every $s_l \to s_{l+1}$ is also an E-step for $0 \le l \le |W_i| - 1$.*

Proof. The only transition during which the environment variables of the composition change is the transition $s_k \to^e s_{k+1}$. Therefore, since $s = s_0$ and $s' = s_m$ satisfy E, and since the actions $\{\alpha_j\}_{j \in W_i}$, and therefore $\{\beta_l\}$, do not change the environment variables, we conclude that all states $\{s_l\}$ satisfy E. We conclude that every transition $s_l \to s_{l+1}$ is an E-step. □

Theorem 1. *Let (S_0, f_0), (S_1, f_1), ..., and (S_{n-1}, f_{n-1}) be compatible stabilizing systems, and $S = \|_{0 \le i < n} S_i$ their composition. If S has a metastable failure, then it has a metastable fault.*

Proof. We prove this theorem by proving its contrapositive: if S has no metastable faults, then S and any compatible environment E satisfy $E \rightsquigarrow^+ \bigwedge_{0 \le i < n} f_i = 0$, which implies $\Box E \Rightarrow \Diamond \Box \bigwedge_{0 \le i < n} f_i = 0$, *i.e.*, S does not have a metastable failure. We proceed by induction on n.

Base case. If $n = 1$, then we have a single component (S_0, f_0). Stabilization for S implies the existence of an environment predicate E such that $E \rightsquigarrow^+ f_0 = 0$; therefore, a compatible environment for S_0 exists, and for any such environment E, based on compatibility we have $E \rightsquigarrow^+ f_0 = 0$ as S_0 has no writing neighbors. This is exactly what we want to prove.

Hypothesis. Any compatible environment E' for the composition $S' = \|_{0 \le i < k} S'_i$ of any $k < n$ compatible stabilizing systems (S'_0, f'_0), (S'_1, f'_1), ..., and (S'_{k-1}, f'_{k-1}), that has no metastable faults, in tandem with S', satisfies $E' \rightsquigarrow^+ \bigwedge_{0 \le i < k} f'_i = 0$.

Step. Let E be a compatible environment for S. Since S has no metastable faults, then there exists some system S_j such that either $W_j = \emptyset$—Let J' be the index set of all such systems—or for any set of actions $\{\alpha_k\}_{k \in W_j}$ with $\alpha_k \in \mathcal{A}_k$ for $k \in W_j$, the actions $\{\alpha_k\}$ are not destabilizing at s, which means that each action can only decrease f_j if positive or maintain it at zero during any E-step; let J'' be the index set of components with the latter quality. For every $j' \in J'$, compatibility implies $E \rightsquigarrow^+ f_{j'} = 0$. Now pick one $j'' \in J''$. Since $f_{j''}$ is a potential function for $S_{j''}$, then $S_{j''}$'s own actions do not increase $f_{j''}$. Now, consider an E-step $s \to s'$ of S where the writing neighbors of $S_{j''}$ take actions $\{\alpha_l\}_{l \in W_{j''}}$—at least one such component takes an action in each transition per our model. This transition is equivalent to some serial order, and according to Lemma 3, all the corresponding intermediate transitions are also E-steps. Therefore, since each action α_l for $l \in W_{j''}$ either decreases $f_{j''}$ if positive or maintains it at zero during an E-step, we conclude that either $f_{j''}(s) > f_{j''}(s')$ or $f_{j''}(s) = f_{j''}(s') = 0$. Therefore, $f_{j''}$ will eventually decrease to 0 and remain there, *i.e.*, $E \rightsquigarrow^+ f_{j''} = 0$. Letting $J = J' \cup J''$, we have shown that $E \rightsquigarrow^+ \bigwedge_{j \in J} f_j = 0$. Consider now the systems with indices not in J. Based on Lemma 2, $E \wedge (\bigwedge_{j \in J} f_j = 0)$ is a compatible environment for the composition of these systems. Moreover, their composition inherits not having a metastable fault from S, as otherwise S would have a metastable fault. Therefore, based on the induction hypothesis, we have $E \wedge (\bigwedge_{j \in J} f_j = 0) \rightsquigarrow^+ \bigwedge_{j \in [n] \setminus J} f_j = 0$. Since we also have $E \rightsquigarrow^+ \bigwedge_{j \in J} f_j = 0$, based on Lemma 1 we deduce:

$$E \rightsquigarrow^+ \Big(\bigwedge_{j \in J} f_j = 0\Big) \wedge \Big(\bigwedge_{j \in [n] \setminus J} f_j = 0\Big),$$

which is just $E \rightsquigarrow^+ \bigwedge_{0 \le j < n} f_j = 0$. This finishes our proof.

□

B The Look-Aside Cache Incident

Following the original implementation demonstrating this failure [26, 35], consider a cache, a webserver, and a database. The webserver receives client requests and forwards them to the cache. If the result is a hit, the webserver responds to the client. If the cache query returns with a miss, the webserver forwards the request to the database, and also starts a timer. If it receives a response from the database before the timer expires, it sends the response to the client *and* updates the cache with the result; if, on the other hand, the result takes too long to come back, it drops the request.

The cache, if full, helps respond to a lot of the client requests, and therefore moderates significantly the load on the database. If a shock (*e.g.*, crash) leaves the cache empty, then suddenly all of the client requests will be forwarded to the database, severely congesting it. As a result, requests will eventually timeout at the webserver, which means that it will not update the cache by virtue of dropping the requests. This behavior stops the cache from adequately warming up after the shock, sustaining therefore the congestion.

In the following, we first apply our characterization to this incident to reveal the faults and the failure. We will then demonstrate that, for this particular incident, removing the fault is not particularly out of reach, and show how simple design decisions help the designer get rid of the fault. Finally, we apply the methodology from §6 to render the system MFT without eliminating the faults.

Potential Functions. The potential function for the cache is $f_1 = C_{max} - C$, the number of keys for which it does not have a value—C_{max} is the maximum number of keys the cache can store values for. It stabilizes in the presence of an environment that populates it with new keys. The potential function f_2 for the webserver is the number of pending requests, and it stabilizes in the presence of an environment that gives it enough acknowledgments and not too many new requests. The potential function for the database is $f_3 = \max\{0, Q - s\}$, where Q is the number of requests in its queue and s is its service rate. It stabilizes in the presence of an environment that does not overload it.

Compatibility. Let E denote an environment that, at each step, gives to the webserver r client requests, where $0 < r < s$. The database's only writing neighbor is the webserver. Because the client load does not overload the database, eventually the database will eliminate any existing congestion and respond to the webserver in a timely fashion, regardless of the webserver's potential. Therefore, we have $E \wedge f_2 = 0 \rightsquigarrow^+ f_3 = 0$. The webserver's writing neighbors are the cache and the database. If the cache is full, and the database is not congested, the webserver will get timely responses for all of the client requests it receives, *i.e.*, we have $E \wedge f_1 = 0 \wedge f_3 = 0 \rightsquigarrow^+ f_2 = 0$. Finally, the cache's only writing neighbor is the webserver. If we repeatedly have $f_2 = 0$, it means that the webserver has no pending requests, implying that the database is not congested; the cache will be populated, and we have $E \wedge f_2 = 0 \rightsquigarrow^+ f_1 = 0$. We have established compatibility.

Metastable Fault. We model the cache with the single action `cache-serve`, with which it responds to the requests sent from the webserver in a step. The cache might be empty, and its response to all of the requests can be a miss, therefore this action can possibly maintain the webserver's potential f_2. We model the webserver with one action as well: the action `webserver-serve`. This action sends new requests to the cache, missed ones to the database, updates the cache, and drops also the requests that time out. This action might send the cache zero updates if all pending requests time out,

and therefore maintains the cache's potential f_1; there exists a metastable fault between the cache and the webserver.

The action `webserver-serve` can increase the database's potential f_3 if it sends the database more than s client requests, and the database's only action `database-serve`, which serves client requests in its queue and responds to the webserver, might not decrease the webserver's potential if all of the corresponding requests have timed out. Therefore, a metastable fault exists between the webserver and the database as well.

Metastable Failure. The two metastable faults are fanned into a metastable failure by two scheduling decisions: (*i*) the webserver schedules to drop requests every T steps for some T, and (*ii*) the database does not prioritize new requests over older ones by simply serving requests from its queue according to arrival order. After the shock that empties the cache, the webserver congests the database by repeatedly sending it more client requests than it can handle. Once this condition persists long enough, old requests will time out at the webserver. Then, since the backlog in the database's queue has only been increasing, *all younger requests* will also eventually time out as they will wait even longer than the first request that timed out. The result is a stable backlog that starves the cache, and therefore sustains itself indefinitely.

B.1 Removing the Faults

Unlike the other case studies, where the fault is irremovable (retry storm) or too delicate to locate a priori (oscillating membership), in this incident formalizing the faults reveals a fix to remove them altogether. The webserver should randomly tag some of the pending requests among each s requests that it sends to the database, and *never time them out.* This way, assuming a suitable distribution of keys in the incoming client requests, the webserver will always update the cache after a shock, and therefore its action will always decrease the cache's potential f_1. This breaks the cyclic destabilizing interaction between the cache and the webserver. The fixed system will never experience the metastable failure outlined above.

B.2 Applying the MFT Methodology

One can render the system MFT without removing the fault.

Derive metastability skeleton. The original implementation showcasing this failure [26] requires a careful design including a memcached cache [4], an NGINX [6] webserver expressed in PHP [7], and a mySQL database [5]. We instead express each component as a Nyx agent, and abstract away every detail that is not relevant to the metastable failure. What remains is the metastability skeleton of the composition, which captures how requests flow in the system and how each agent responds to different types of requests. We ignore key replacement in the cache, as it does not pertain to the warmup phase and has little effect on this particular failure.

Study dynamics. Once equipped with the metastability skeleton of the system, we can probe the system's behavior by asking Nyx to generate the vector field. For simplicity, we only look at the vector field of cache and database potentials. Figure 5 demonstrates the result, where the client sends requests at a rate of $r = 40$ requests per step, the database serves them at a rate of $s = 10$ requests per step, and the webserver timeout is $T = 8$ steps. We instruct Nyx to generate 100 traces, using two simultaneous shocks with a random duration: cache crash and surge in client demand.

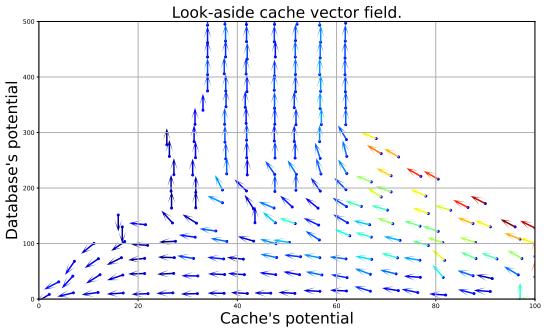


Figure 5. The Nyx vector field of the look-aside cache. Arrow color indicates tendency intensity at that point; warmer means higher intensity.

We make two observations: (*i*) the figure demonstrates the two coexisting tendencies, one pulling towards a full cache and an empty database, and the other pulling towards a stagnant cache and an ever-increasing database backlog, and (*ii*) the backlog is stable; once the bad tendency wins the tug of war, there is no hope for the system to recover without manual intervention. This reinforces our account of the failure, wherein all pending requests eventually time out once the first timeout happens.

Manage scheduling. To make this system MFT, we have to change how the components schedule their actions. One option is to pick a T large enough that the cache has ample time to warm up. Another, more subtle approach is to change the scheduling in the database, and enforce a mechanism for prioritizing younger requests. This reduces the chances of them timing out, and gives the system more time to fill the cache. The details of the prioritization depend on the parameters and the key distribution in client requests.

Complete the proof. Starting from any arbitrary state in the aftermath of a shock emptying the cache, a race starts between the tendency filling in the cache and the timer ticking in the webserver. If the cache is sufficiently populated before the first timeout in the webserver, the system will never experience a metastable failure. On the other hand, if the first timeout happens before the cache is sufficiently populated, then the cache will never be sufficiently populated. Therefore, proving correctness for this system entails proving that the cache fills fast enough for there to be no timeouts, which corresponds to proving R2. As for R1, since, as explained

above, being sufficiently full is a stable property, the system will never timeout, and therefore never destabilize itself.