It is OK to be Metastable

Aleksey Charapko

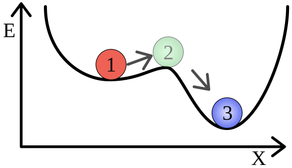
Department of Computer Science
University of New Hampshire

March 20, 2024

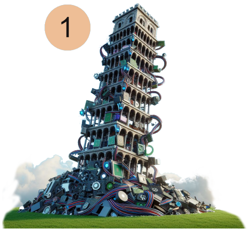
University of New Hampshire

Table of Contents

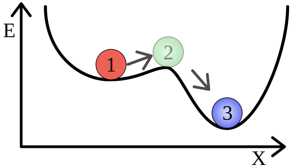
Metastable Failures



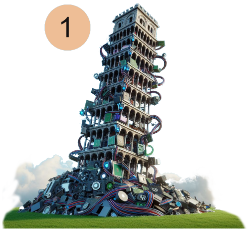
1. Everything is normal



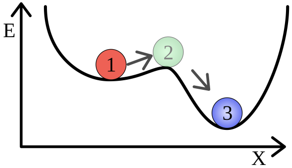
Metastable Failures



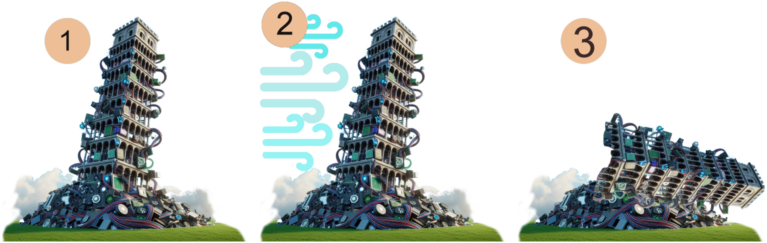
1. Everything is normal
2. A trigger sends the system "over the hump"



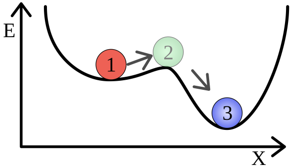
Metastable Failures



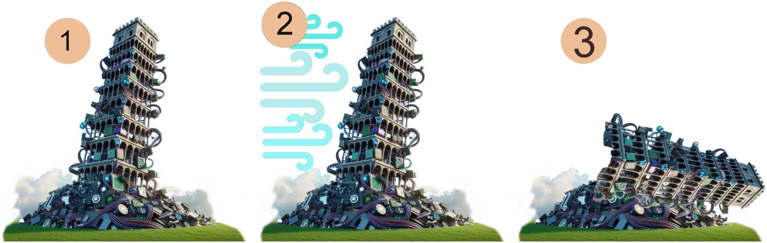
1. Everything is normal
2. A trigger sends the system “over the hump”
3. Everything is not normal



Metastable Failures

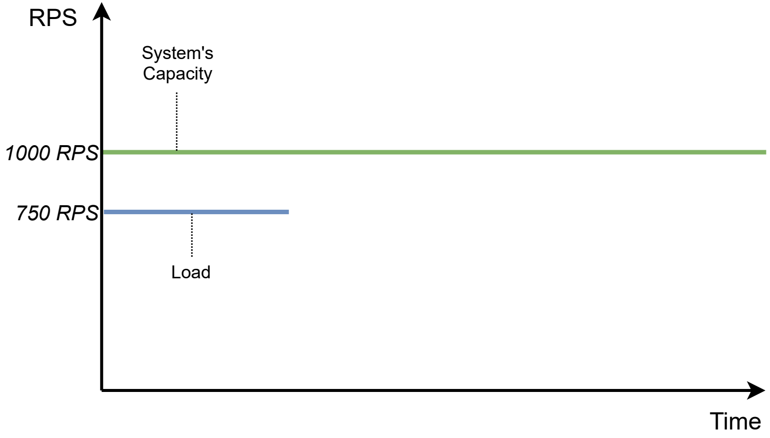


1. Everything is normal
2. A trigger sends the system "over the hump"
3. Everything is not normal
4. Some mechanism keeps everything not normal even after the trigger is gone



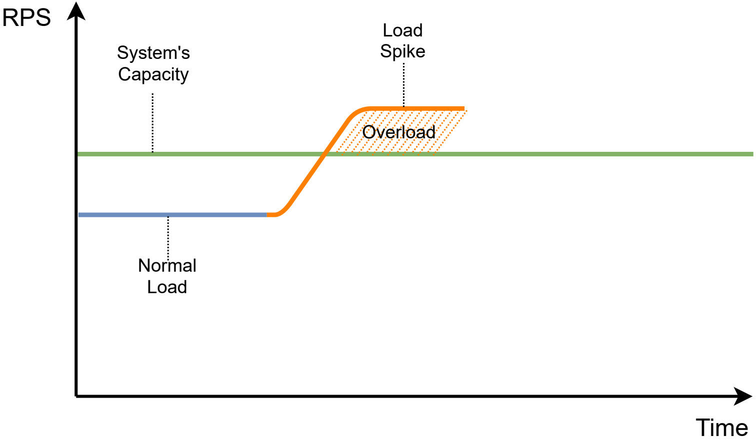
Hypothetical Example: Normal Operation

- Hypothetical system with 1000 RPS of capacity
- Normal load of 750 RPS
- In Metastable *vulnerable* state



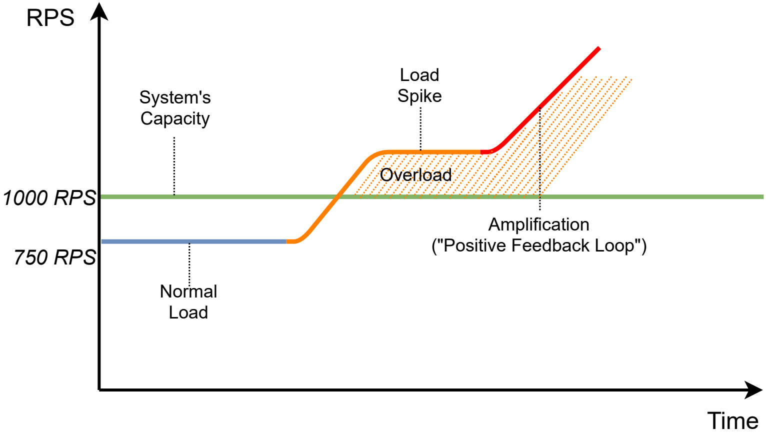
Hypothetical Example: Trigger

- Some **trigger** creates an overload
  - HW or SW failure
  - *Expectation* failure



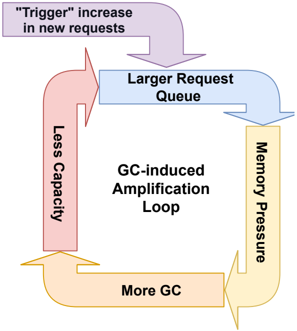
Hypothetical Example: Amplification

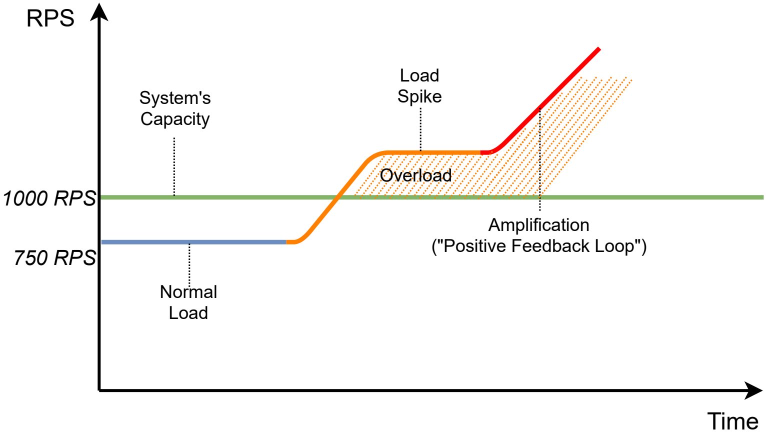
- Some hidden mechanism **amplifies** the overload



Hypothetical Example: Amplification

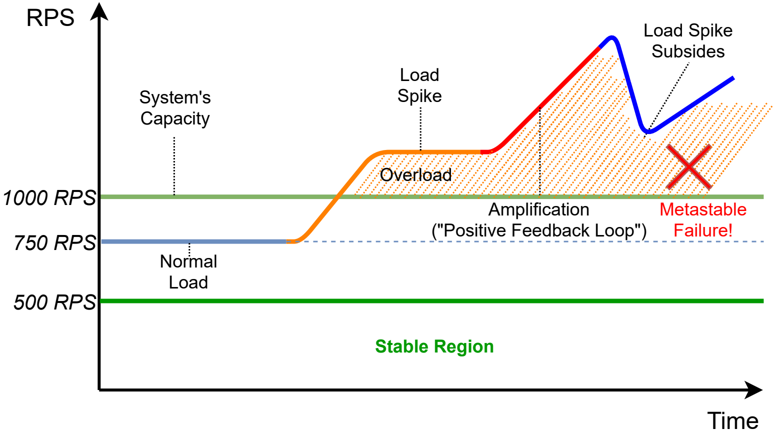
- Some hidden mechanism **amplifies** the overload





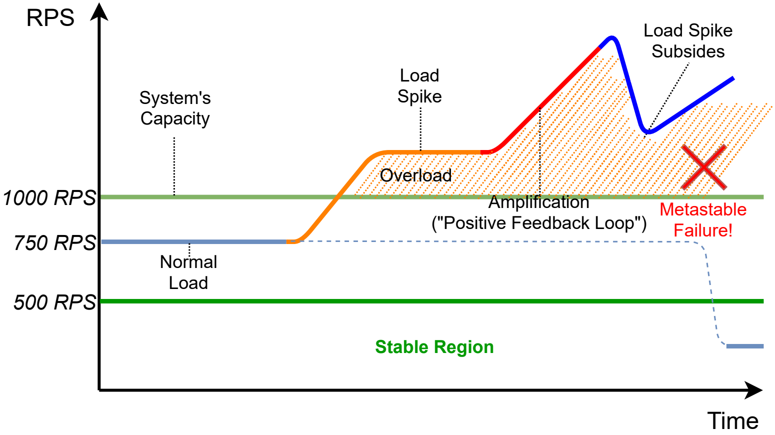
Hypothetical Example: Metastable Failure

- Trigger is fixed, but the system is still overloaded

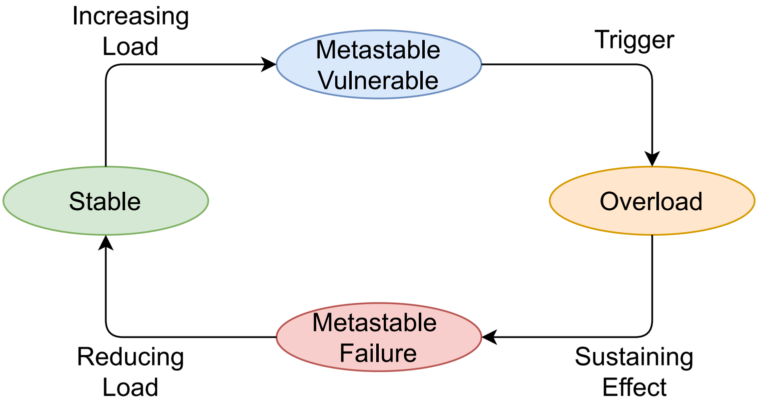


Hypothetical Example: Getting out of Metastable Failure

- Reduce organic offered "normal" load below some *stable* threshold



Metastable Failure Life-cycle



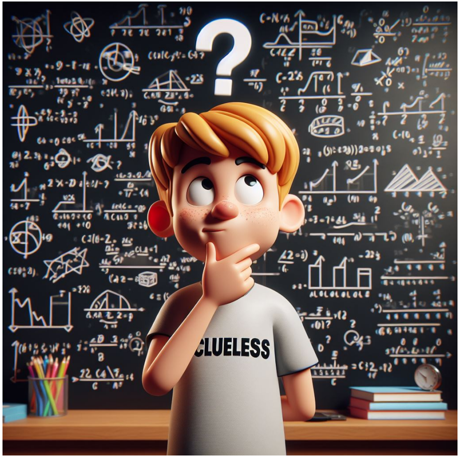
Being Metastable is OK

- Three pillars of being Metastable:
    1. understanding the environments, algorithms, and workloads.
    2. trigger-resistant design
    3. protection of vulnerable components

Table of Contents

University of New Hampshire

"Knowledge is Power"



With knowledge, we can avoid "expectation failures"

Understanding Environments

Expectation failures arise from a mismatch between the environment's capabilities and the system's needs.

Case Study: Cloud Latency

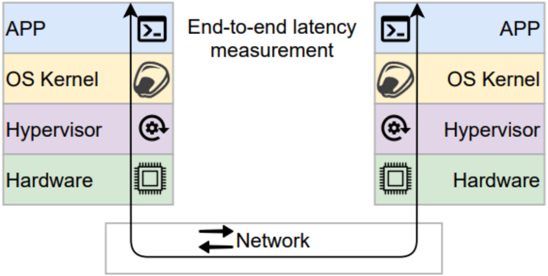
Cloud is complicated – shared resources, "noisy neighbor syndrome," etc.

- Knowing how well a cloud performs is crucial for configuring systems to run in the cloud.

Case Study: Cloud Latency

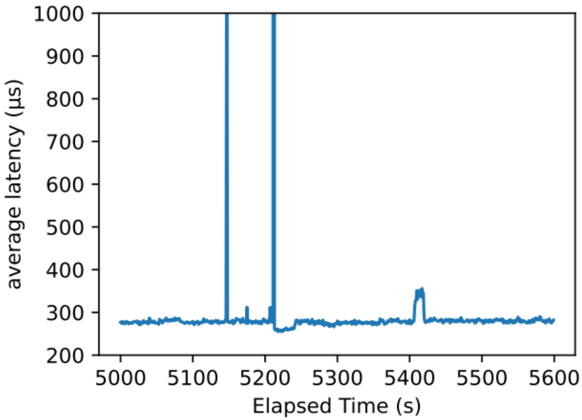
Cloud is complicated – shared resources, "noisy neighbor syndrome," etc.

- Knowing how well a cloud performs is crucial for configuring systems to run in the cloud.
- Let's look at communication latency between nodes



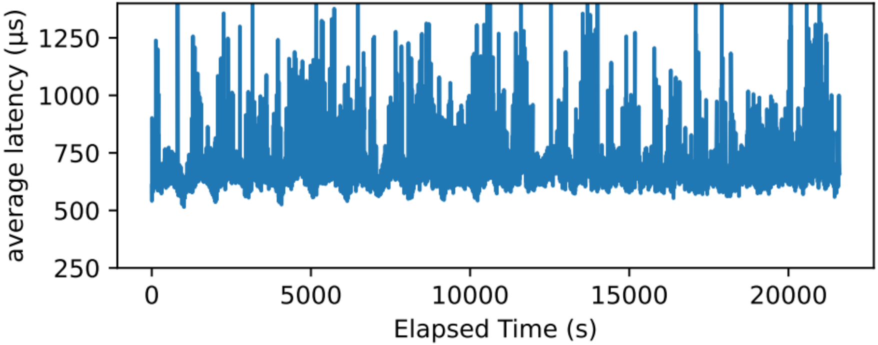
Case Study: Cloud Latency Example

Would you expect spikes $3,000\times$ over the median latency?



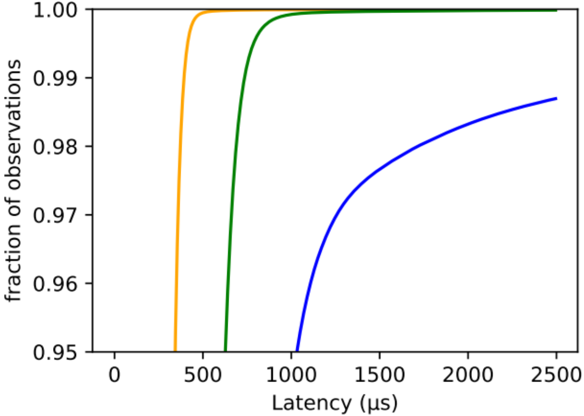
Case Study: Cloud Latency Example

Would you expect lots of variation and 20-minute cycles?



Case Study: Cloud Latency Example

Would you expect very high tail latency?



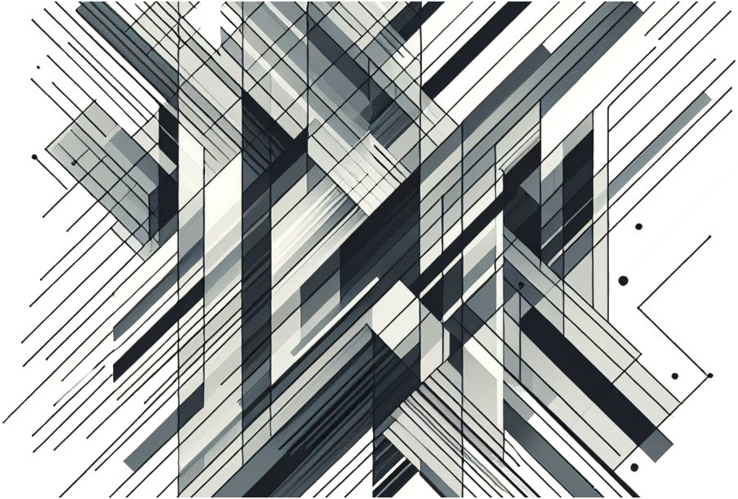
University of New Hampshire

Latency Expectations Mismatch

Cloud	Cloud #1			Cloud #2		
	Same Subnet	Cross Subnet	Cross Az	Same Subnet	Cross Subnet	Cross Az
median (μs)	270.0	260.0	555.0	595.0	590.0	1310.0
p5 (μs)	215.0	185.0	445.0	390.0	395.0	755.0
p25 (μs)	245.0	230.0	490.0	485.0	485.0	925.0
mean (μs)	283.0	279.0	555.4	696.7	661.9	1339.5
p90 (μs)	325.0	325.0	645.0	890.0	875.0	1665.0
p95 (μs)	345.0	345.0	670.0	1035.0	995.0	1810.0
p99 (μs)	395.0	395.0	755.0	3135.0	1845.0	4865.0
p999 (μs)	470.0	470.0	1105.0	9795.0	8925.0	16670.0
p9999 (μs)	760.0	745.0	1905.0	21200.0	21705.0	23085.0

- A system with expectations for low latency may work better in cloud #1 than cloud #2

Understanding Algorithms



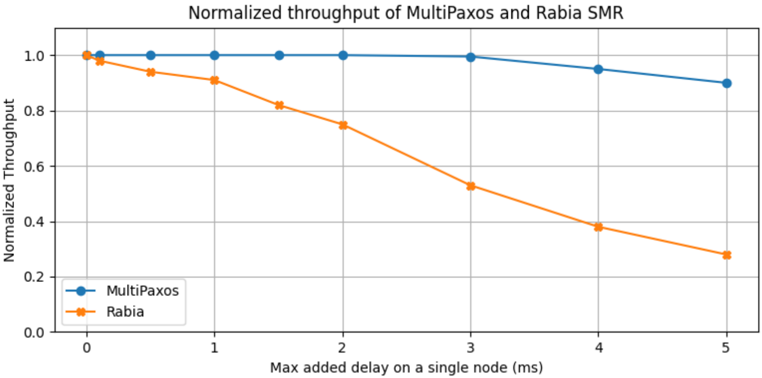
University of New Hampshire

Case Study: State Machine Replication

State Machine Replication (SMR) is a very common class of algorithms used in storage and configuration systems.

- Some algorithms perform well under networks with unreliable latency
- And some expend resources when bad communication timing throws them off the "common case"

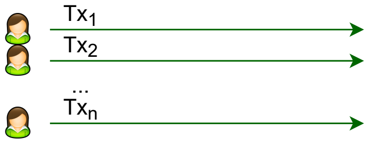
Case Study: SMR – MultiPaxos & Rabia



The difference in performance is due to environment expectations in Rabia – it needs timely delivery of messages to nodes!

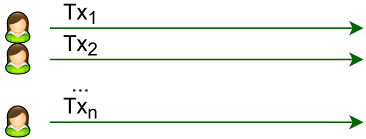
Case Study: Transactions

- In the common case, most concurrent transactions have no contention

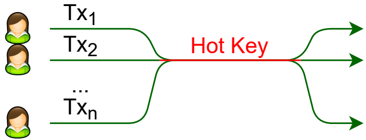


Case Study: Transactions

- In the common case, most concurrent transactions have no contention

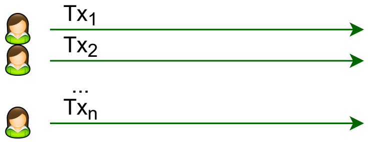


- But what if we have a "hot" shared object or key?

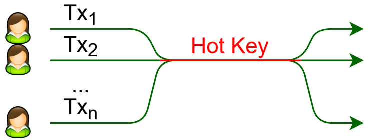


Case Study: Transactions

- In the common case, most concurrent transactions have no contention



- But what if we have a “hot” shared object or key?



- Transaction aborts and/or lock contention

Understanding Workloads

- Sometimes we can pick or configure algorithms (next section!) to match the environment better.
- But in other cases, like transactions, we may be out of luck.
- So we need to understand workload behaviors that may cause algorithms/systems to "trigger."

Case Study: Coordinated Clients

1. A database works perfectly fine
2. Some code at the client side of the application runs on the timer once a day

Case Study: Coordinated Clients

1. A database works perfectly fine
2. Some code at the client side of the application runs on the timer once a day
3. On all active clients at the same time

Case Study: Coordinated Clients

1. A database works perfectly fine
2. Some code at the client side of the application runs on the timer once a day
3. On all active clients at the same time
4. This code runs an expensive transaction

Case Study: Coordinated Clients

1. A database works perfectly fine
2. Some code at the client side of the application runs on the timer once a day
3. On all active clients at the same time
4. This code runs an expensive transaction
5. A database stops working

Table of Contents

University of New Hampshire

Trigger Resistance

- While we cannot *avoid* triggers, we may be able to design systems to tolerate *some* common triggers better.

Step 1: Avoiding Expectation Mismatches

This one is pretty straightforward – if we know what to expect from the environment, algorithms, and workloads, we can avoid many expectation mismatches.

Step 1: Avoiding Expectation Mismatches

This one is pretty straightforward – if we know what to expect from the environment, algorithms, and workloads, we can avoid many expectation mismatches.

- Example – aggressive timeouts and flaky network.
  - timeouts may cause false positives on failure detectors
    $- >$ systems undergo unnecessary recoveries
    $- >$ expend resources that could have been used for useful work

Step 2: Designing for Practical Fault-Tolerance

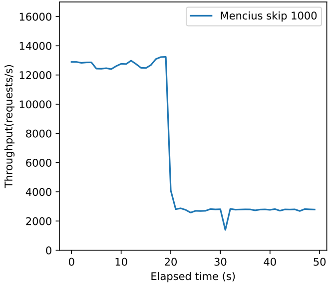
- Many algorithms are designed for fault-tolerance

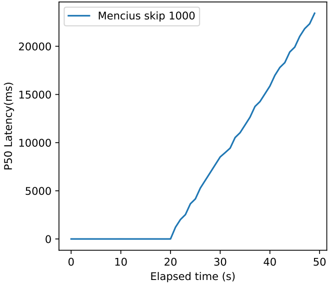
Step 2: Designing for Practical Fault-Tolerance

- Many algorithms are designed for fault-tolerance
- Many are designed by academics...

Step 2: Designing for Practical Fault-Tolerance

- Many algorithms are designed for fault-tolerance
- Many are designed by academics...
- "**algorithmic** fault-tolerance" – a system that can safely tolerate failures, but cannot keep up with the load.



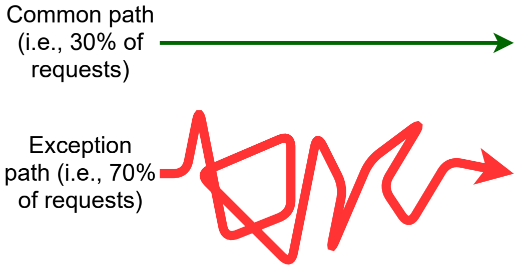


Step 3: Avoiding Overoptimizations on Common Path

Common path (i.e., 90% of requests)

Exception path (i.e., 10% of requests)

Step 3: Avoiding Overoptimizations on Common Path



- Under some conditions (often workload-related), systems may shift to the "exception" path more frequently

Step 4: Workload Engineering

Workloads can impact the algorithms and systems

- Minimizing this impact may require *workload engineering* – designing applications to avoid creating “bad workload” situations for algorithms.
    - A lot of workload engineering focuses on avoiding “hot” keys or objects in parts of systems that do transactional work.

Table of Contents

University of
New Hampshire

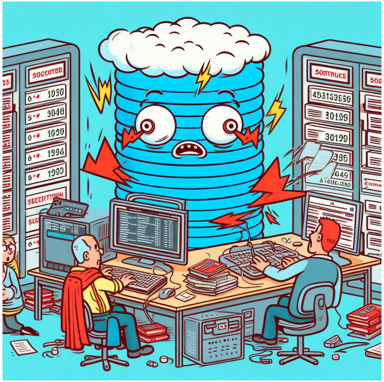
Trigger Resistance is not Enough

- Despite the trigger-resistant design, triggers can still develop into metastable failures

Trigger Resistance is not Enough

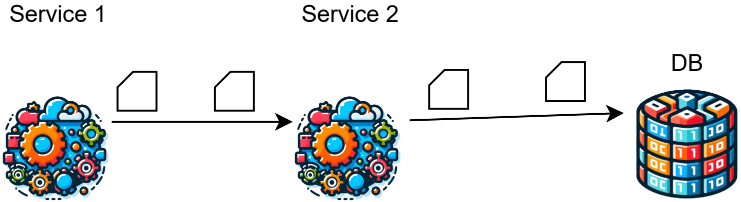
- Despite the trigger-resistant design, triggers can still develop into metastable failures
- Some components of complex systems are more vulnerable
  - We can protect them from failing (at the expense of user experience)

Stateful Components are Vulnerable

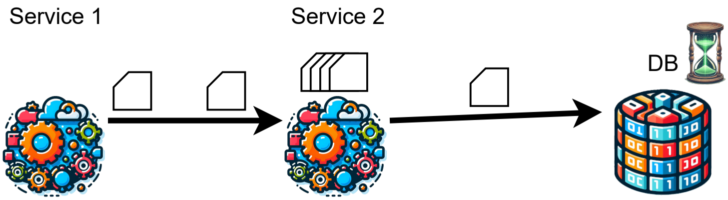


It is harder to quickly scale stateful components compared to stateless services.

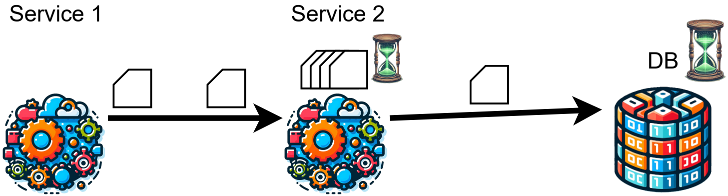
Some Simple Service-Oriented System



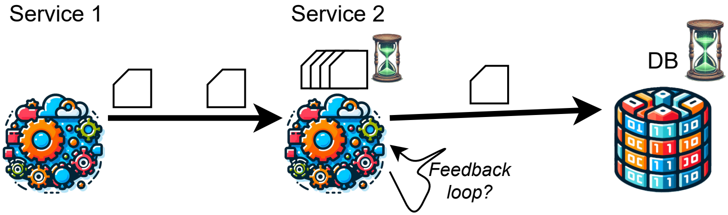
Stateful Component get Overloaded



Overload Propagates Downstream

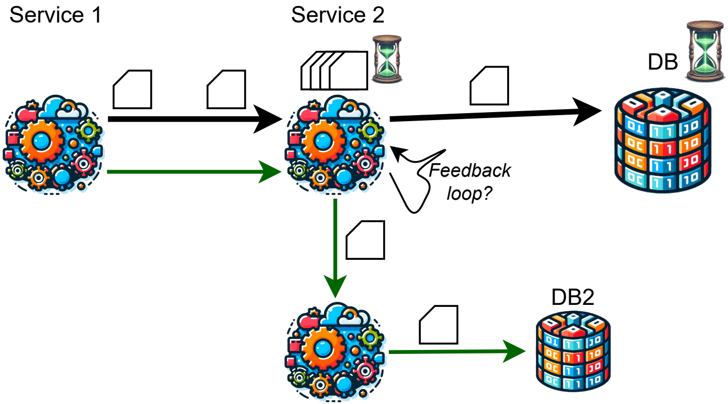


Increasing Danger of Metastable Failures

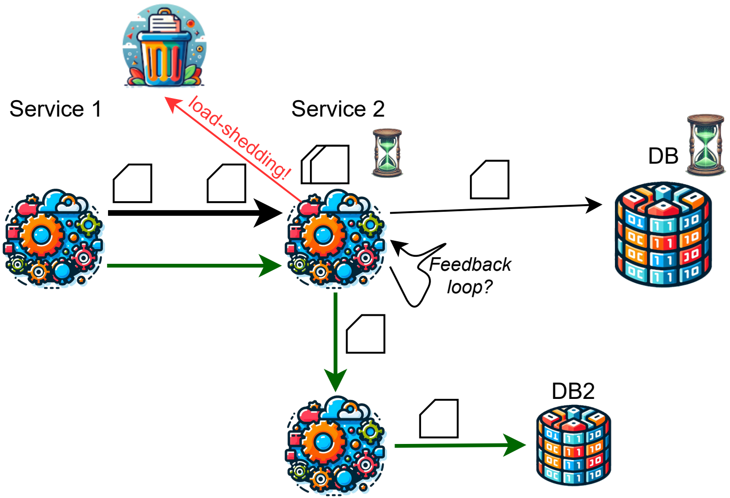


University of New Hampshire

Service Tend to be more Complex

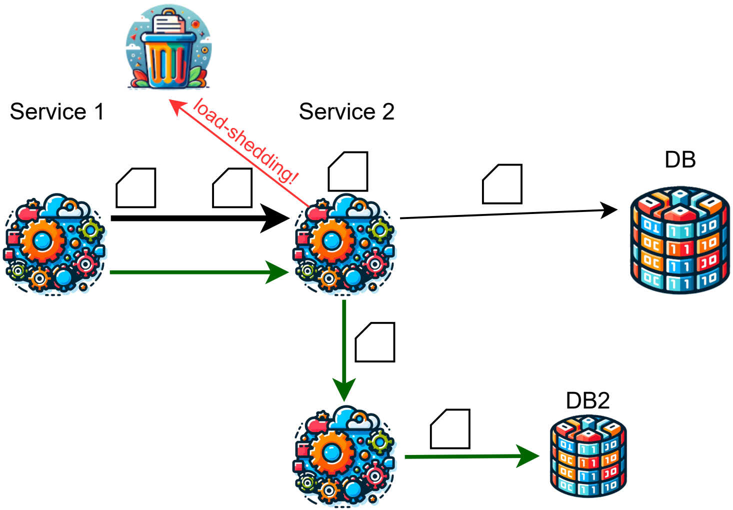


Load-Shedding



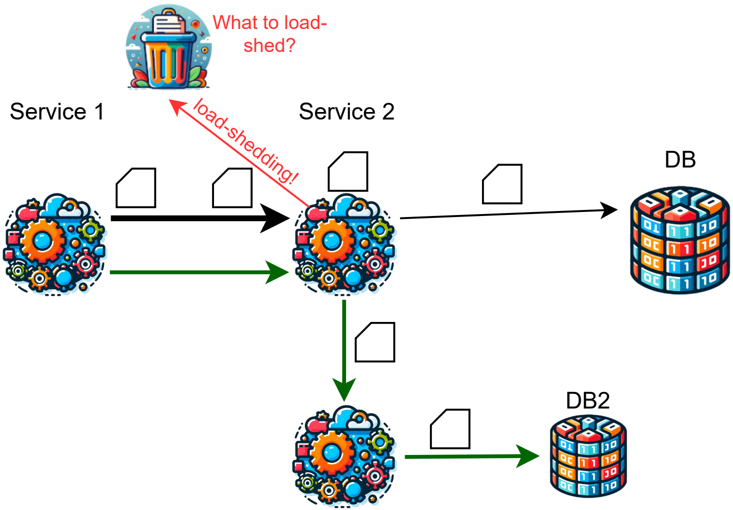
University of New Hampshire

Stateful Component Load Decreases

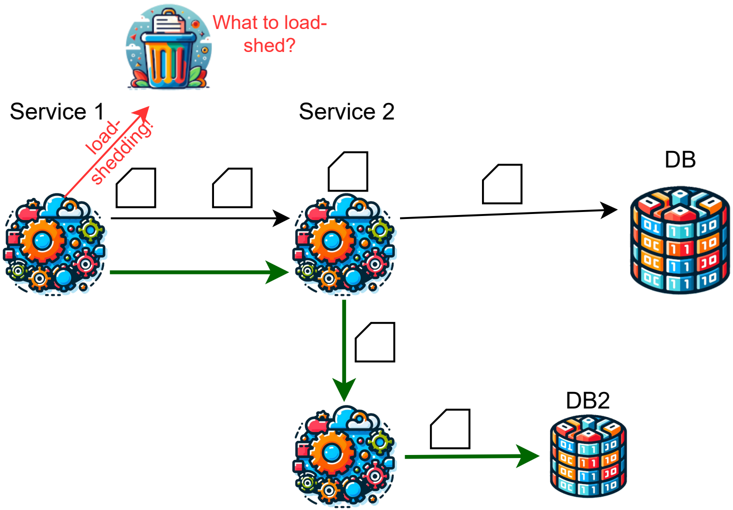


University of New Hampshire

But what to Load-shed?



And when to Load-shed?



Summary

- It is OK to be **Metastable vulnerable**
- Minimize the risks of a **metastable failure**
  - By protecting vulnerable components
  - By practicing trigger-resistant design