One-Size-Fits-None: Understanding and Enhancing Slow-Fault Tolerance in Modern Distributed Systems

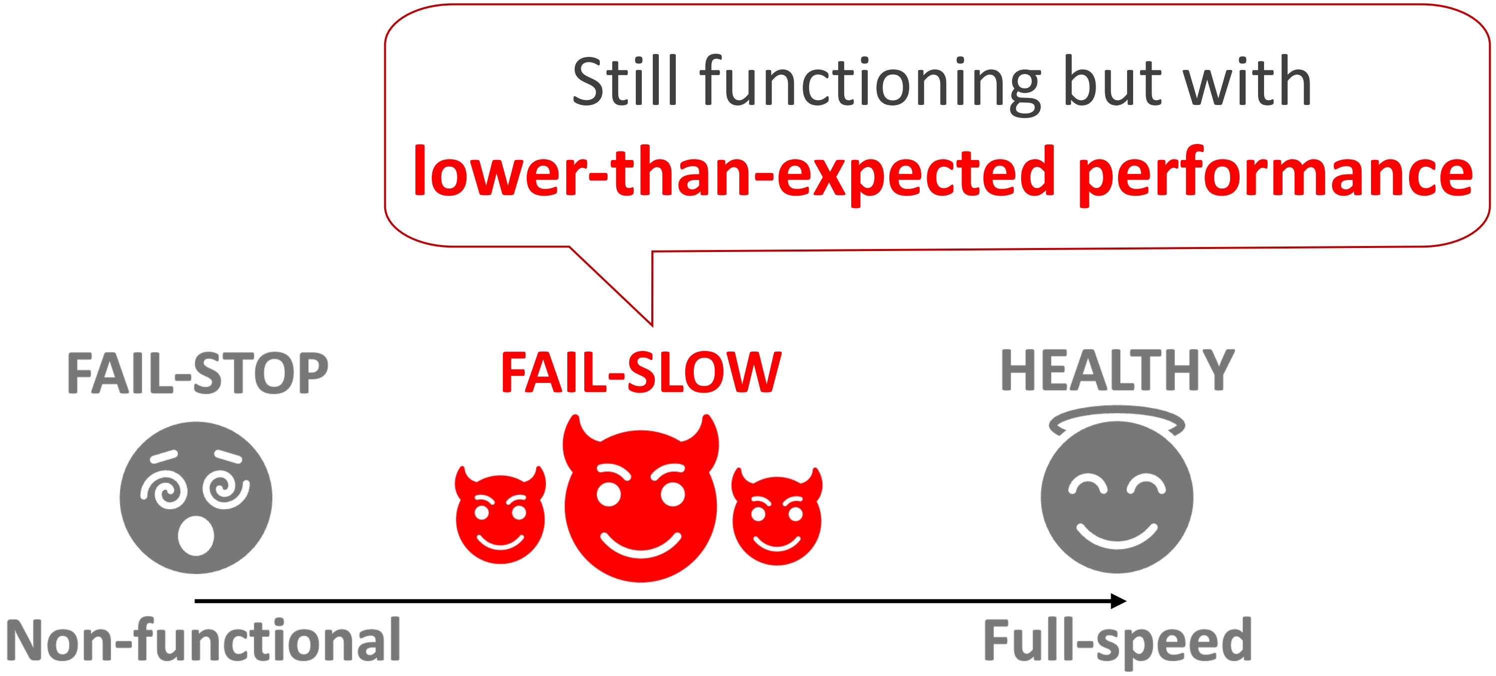
Ruiming Lu, Yunchi Lu, Yuxuan Jiang,
Guangtao Xue, Peng Huang

NSDI ’25

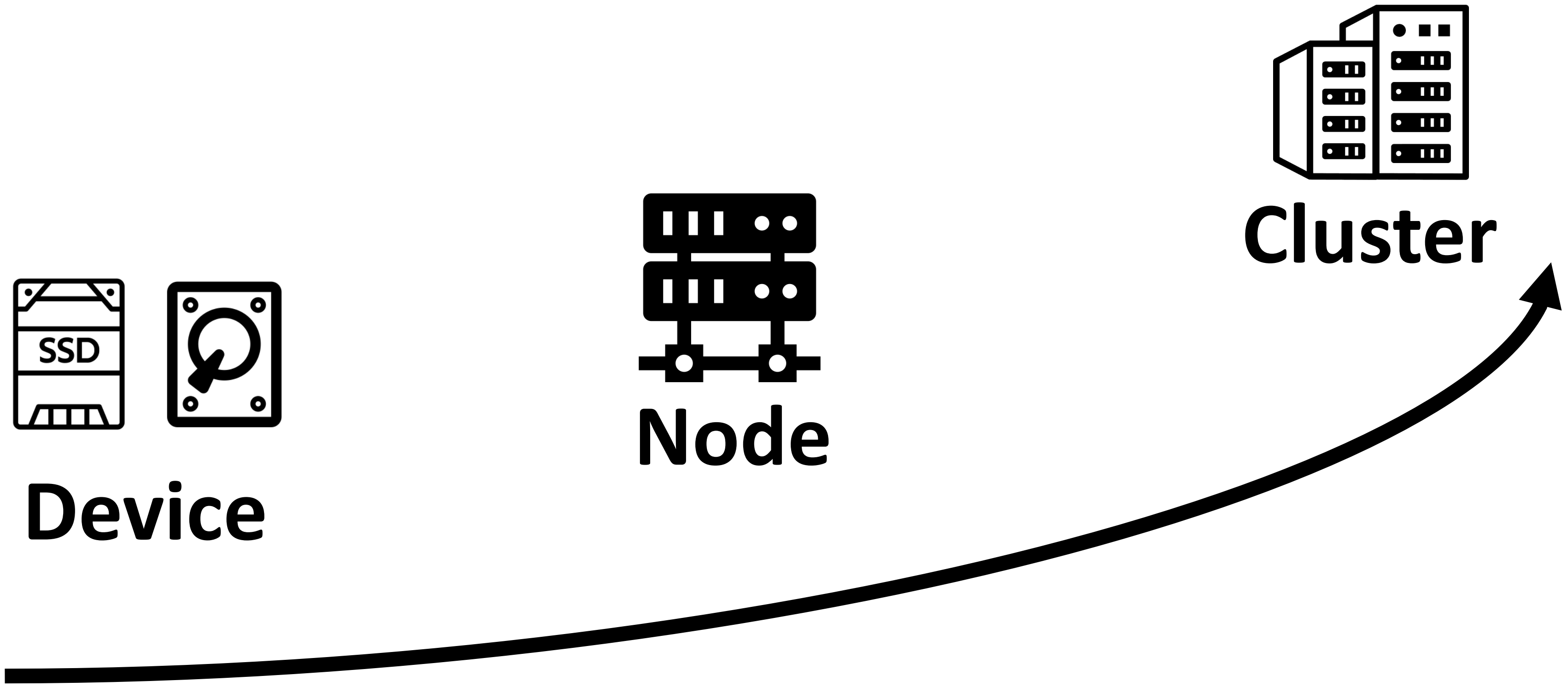
UNIVERSITY OF MICHIGAN

Challenges for distributed system fault tolerance

- **Failures in The Wild**
  - **Fail-Slow**
  - Fail-Stop
  - Metastable
  - …



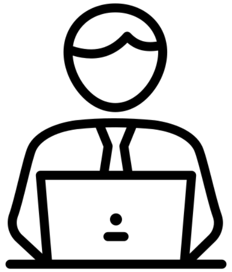
Fail-slow is a severe problem



"**Cascade** to **node-** or even **cluster-**level limplock[1]."

[1] Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems, Do et al.

Fail-slow is **not uncommon**



Annual fail-slow failure rate is **1-2%**[2]!

As frequent as fail-stop incidents!

[2] IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services, Panda et al.

Fail-slow is hard to handle

"System components shall be **either correct or stopped**[3]"

Lucky me! I am in between!

FAIL-SLOW

[3] Gray Failure: The Achilles' Heel of Cloud-Scale Systems, Huang et al.

Slow-fault tolerance studied in 2013

Limplock [SoCC ’13]:

- **Focus on Hardware**
  - Disk and NIC
- **Worst-Case Scenario**
  - Up to **1000**× and persistent slowdown

⚠ **Slow faults are way more complicated!**

varying severity, duration, timing, etc.

Evolvement from 2013 to 2025

- **More Powerful Hardware**
  - Network: 100 Mbps -> 100 Gbps
  - Storage: 600 MB/s -> 6GB/s
  - CPU cores: 4—8 -> ~128
- **Advances in Software Design**
  - Decade’s Bug Fixes
  - Asynchronous Programming
  - Event-Driven Design

Slow-Fault Tolerance in Modern Distributed Systems ?

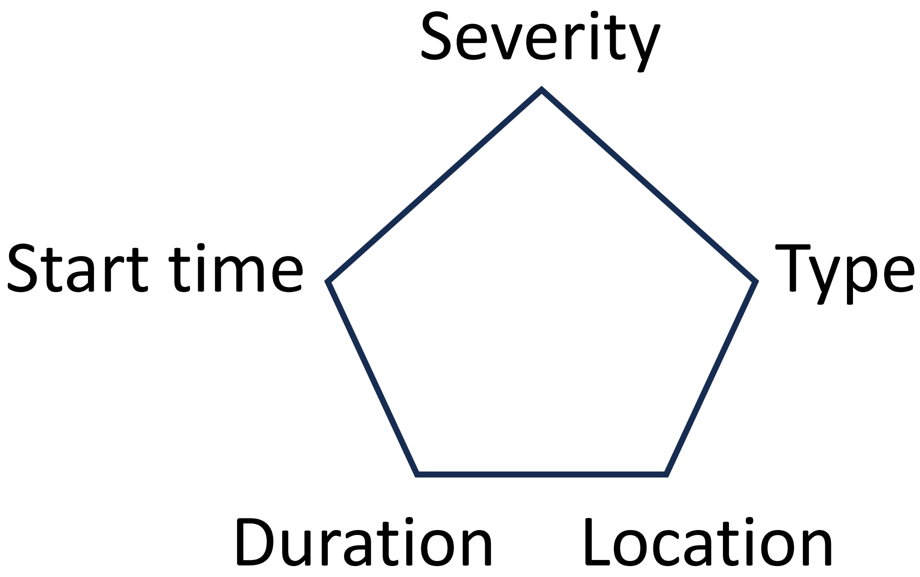
Our studied systems

cassandra | etcd | APACHE HBASE | CockroachDB | hadoop | kafka

- **6 widely-used distributed systems:**
  - Latest stable versions
  - Diverse services:
    - Database, big data, storage, and streaming
  - Tested by cloud benchmarks with distinct workloads
    - e.g., for DB: read-only, write-only, mixed, range query, and transaction

Evaluating slow-fault tolerance is hard

- **Slow faults are multi-faceted**



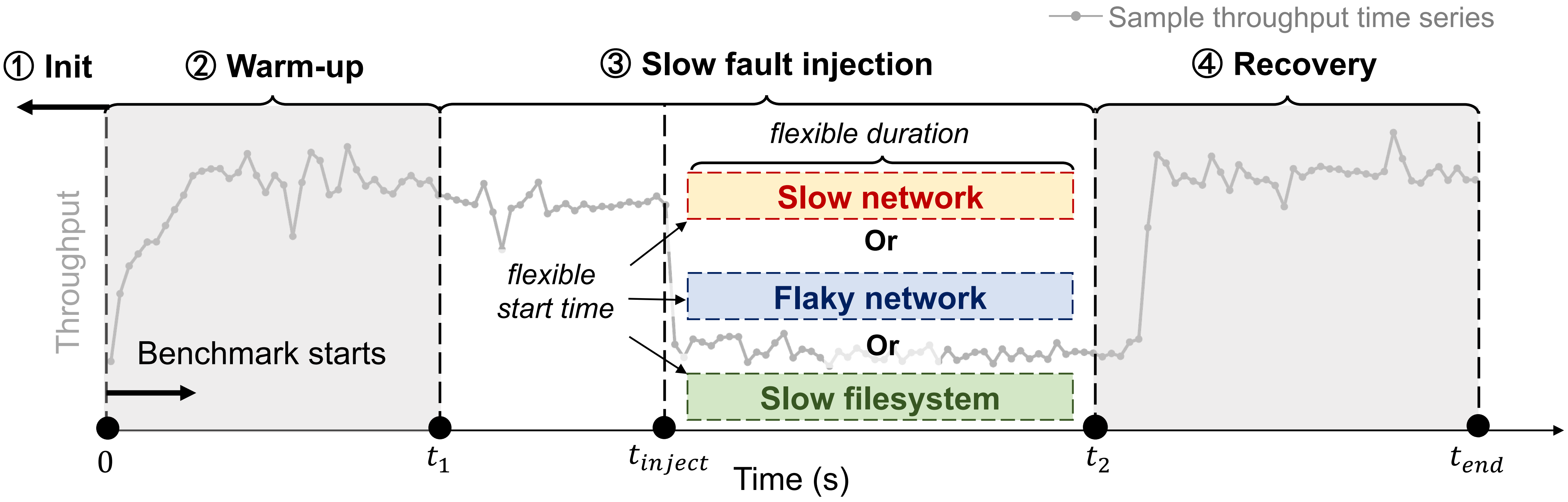
Many combinations to test

Hard to quantify slow-fault tolerance

We propose:

A slow-fault injection testing pipeline

Automated testing



High coverage of slow faults

- Start time
- Duration
- Fault type
- Severity
- Location

We find:

Slow-fault tolerance is highly ***sensitive*** to

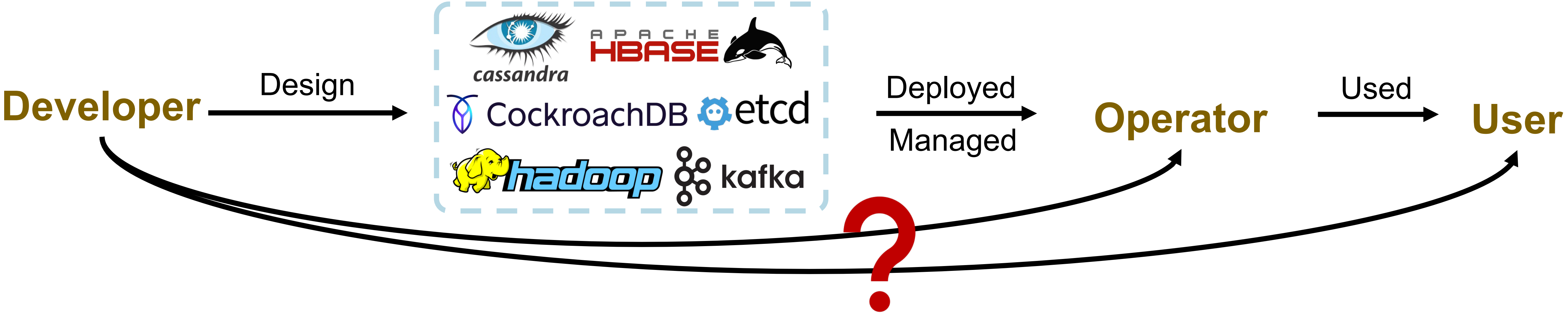
deploying environments and **slow faults**

4 findings **5 findings**

Hard for developers to anticipate future deployment

Developer —Design→ [cassandra, Apache HBase, CockroachDB, etcd, hadoop, kafka] —Deployed / Managed→ **Operator** —Used→ **User**

Hard for developers to anticipate future deployment



Hard for developers to anticipate future deployment

How systems are **deployed**
(e.g., hardware resources, software configs)

Developer cannot anticipate by **Operator** **User**

What workloads are running
(e.g., distinct IO patterns)

We find:

Slow-fault tolerance is highly ***sensitive*** to

Resources **Configs** **Workloads**

We find:

Slow-fault tolerance is highly ***sensitive*** to

Resources **Configs** **Workloads**

Does Tuning Configurations Help?

Slow-related configs

```
hbase.ipc.slow.metric.time
hbase.regionsever.wal.slowsync.ms
hbase.regionserver.wal.roll.on.sync.ms
hbase.regionserver.wal.sync.timeout
hbase.rpc.timeout
hbase.client.retries.number
...
```

7,776 *combinations of configurations*

finetune →

540 *machine hours*

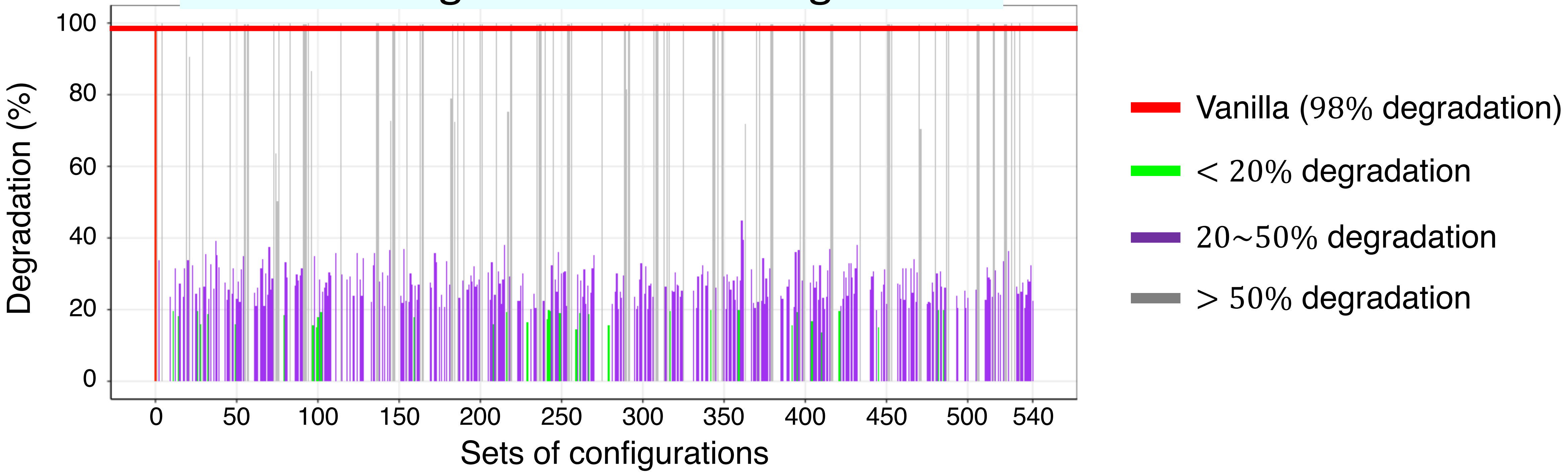
Adapt to **fixed**:
- Slow faults
- Workloads
- Resources

Static setup

Tuning configs under static setups

Under **fixed** slow faults, workloads, and resources:

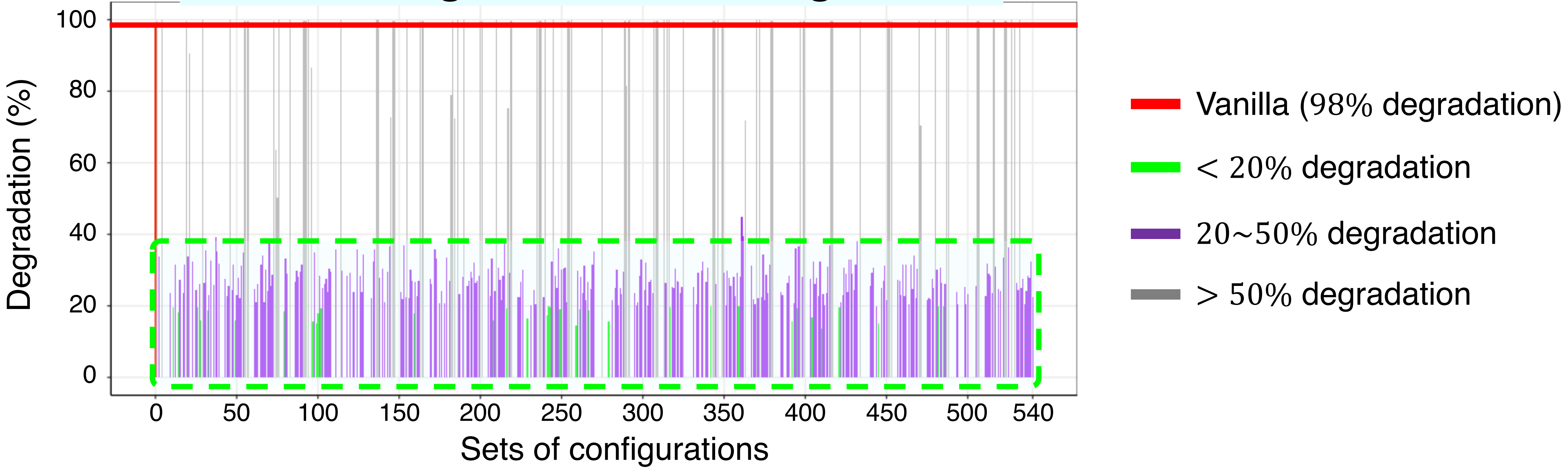
Default configs achieve **98%** degradation



Tuning configs under static setups

Under **fixed** slow faults, workloads, and resources:

Default configs achieve **98%** degradation



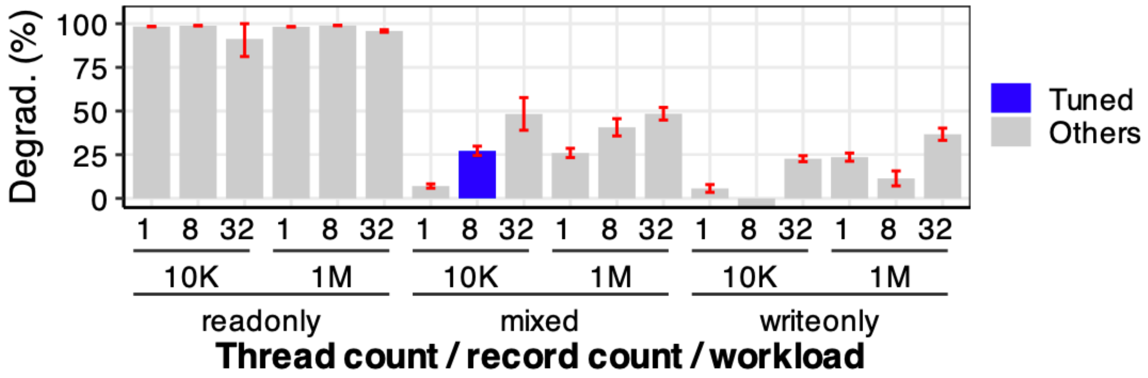
Finetuned configs can get **~20%** degradation

Pick the **optimal** configs under **static** setups

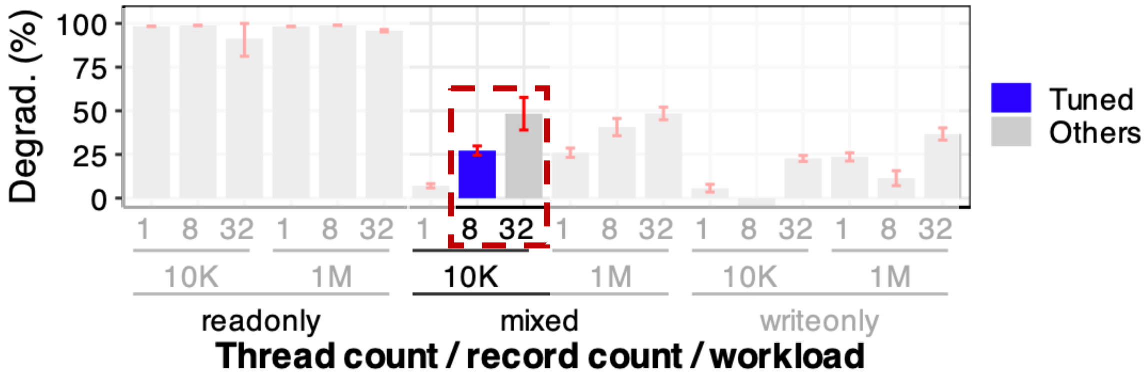
↓

Test under **different** **workloads**

Test under **different workloads**



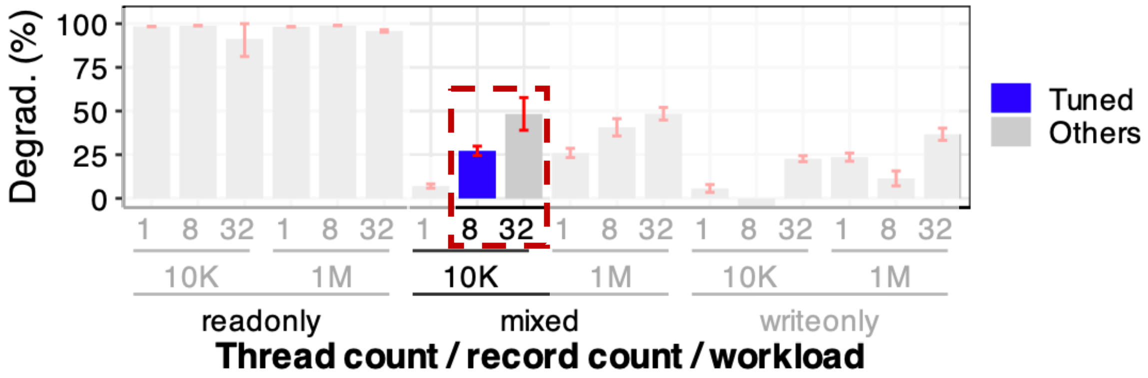
Test under **different workloads**



	8 threads
Setup 1 (fine-tuned)	✓ **27%**
Setup 2 (suboptimal)	30%

Previously optimal setup does not work well

Test under **different** workloads



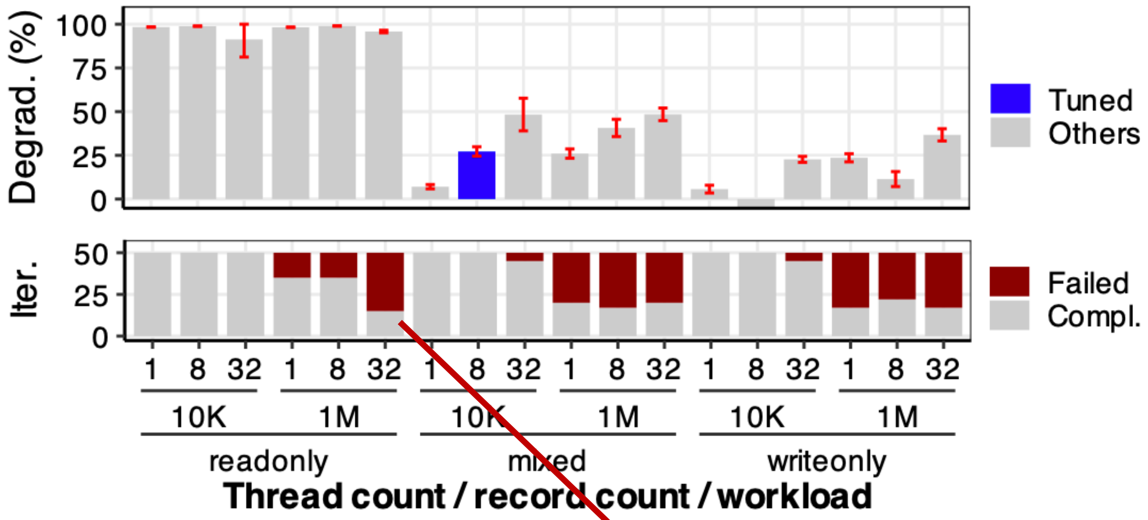
Not always optimal

41% worse when workload changes

	8 threads	32 threads
Setup 1 (fine-tuned)	✓ **27%**	✗ **48%**
Setup 2 (suboptimal)	30%	34%

Previously optimal setup does not work well

Test under **different** **workloads**



35 out of 50 runs failed

Not always optimal

Overfit & harm availability

Our finding:

Tuning configs only improves tolerance under

static, controlled setups

Insight:

Relying on ***static, fine-tuned*** configurations

makes a system's slow-fault tolerance ***fragile***

<More findings in the paper>

Slow-fault tolerance is highly ***sensitive*** to

Resources ~~Configs~~ Workloads

Scaling up resources improves performance but **adversely expands** (up to **10×**) the impact of slow faults

<More findings in the paper>

Slow-fault tolerance is highly ***sensitive*** to

~~Resources~~ ~~Configs~~ **Workloads**

Danger zone commonly exists:

slightly heavier slowness ⇒ **significantly higher** degradation

e.g., in Cassandra: network delay **0.1ms ↗ 1ms** ⇒ degradation **10% ↗ 50%**

We find:

Slow-fault tolerance is highly ***sensitive*** to

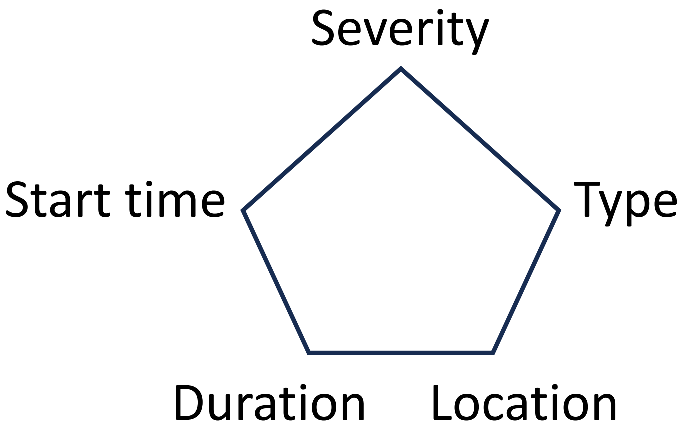
~~**deploying environments**~~ and **slow faults**

4 findings **5 findings**

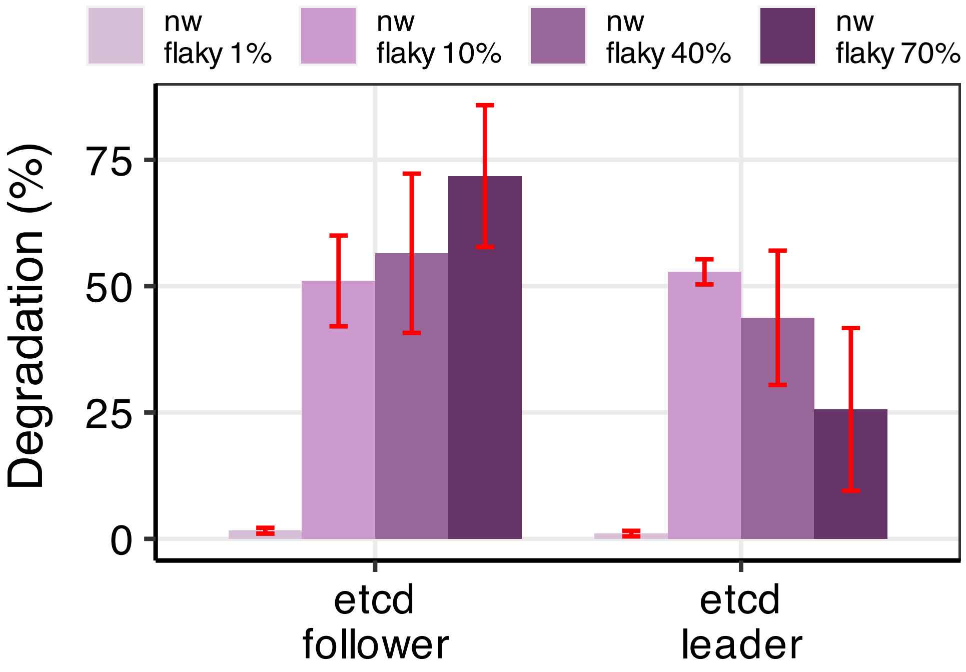
We find:

Slow-fault tolerance is highly ***sensitive*** to

slow faults ← Injection test — testing pipeline



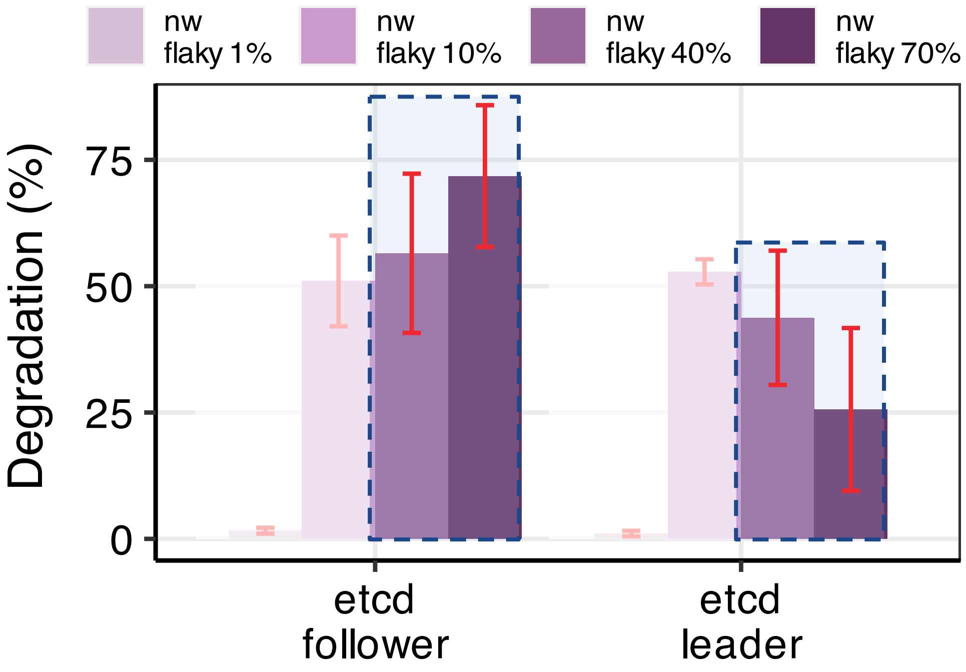
Slow-fault injection test



Severity

Location

Slow-fault injection test



*Compared to a slow **leader**,*
*a slow **follower** yields…*

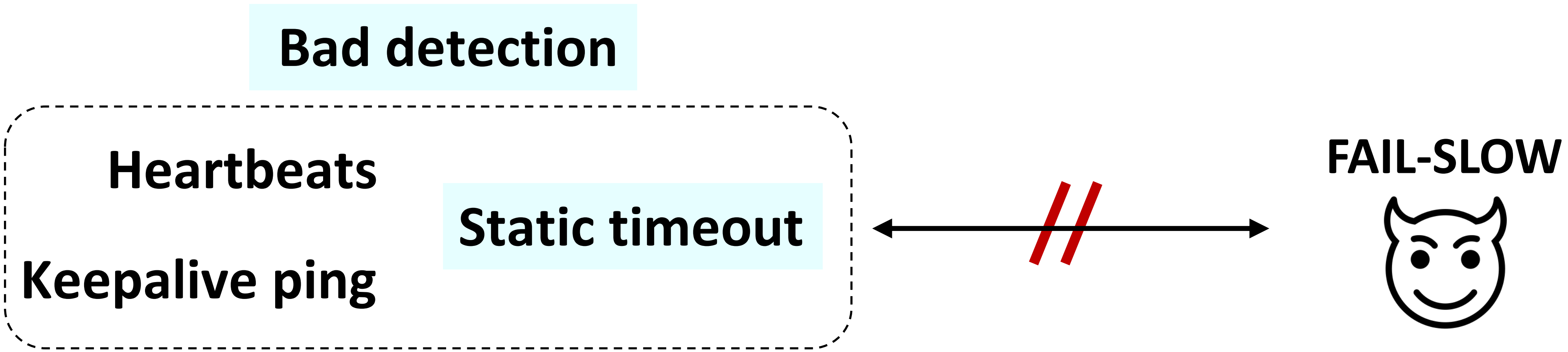
$p40$ **30%** higher degradation

$p70$ **177%** higher degradation

Our finding:

A slow ***follower*** is ***more harmful*** than a slow ***leader***

Static timeout ⟹ Ineffective detection



In practice, how do developers detect slowness?

Static-threshold-based slow detection

Slow sync detection in HBase

```
public void postSync(syncTime) {

}
```

(sync, query, logging)

System metric slow?

Static-threshold-based slow detection

Slow sync detection in HBase

```
public void postSync(syncTime) {
    if (syncTime > 100ms) {

    }
}
```

(sync, query, logging)

System metric slow?

\> warning threshold

Static-threshold-based slow detection

Slow sync detection in HBase

```
public void postSync(syncTime) {
    if (syncTime > 100ms) {
        LOG.INFO(...);
    }
}
```

(sync, query, logging)

System metric slow?

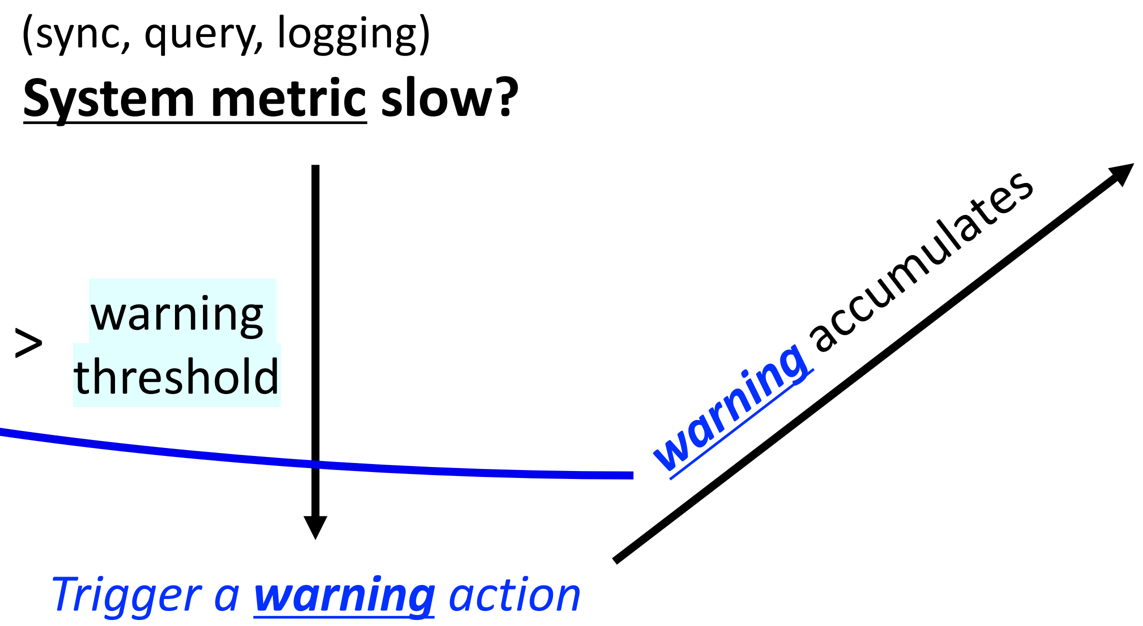
> warning threshold

*Trigger a **warning** action*

Static-threshold-based slow detection

Slow sync detection in HBase

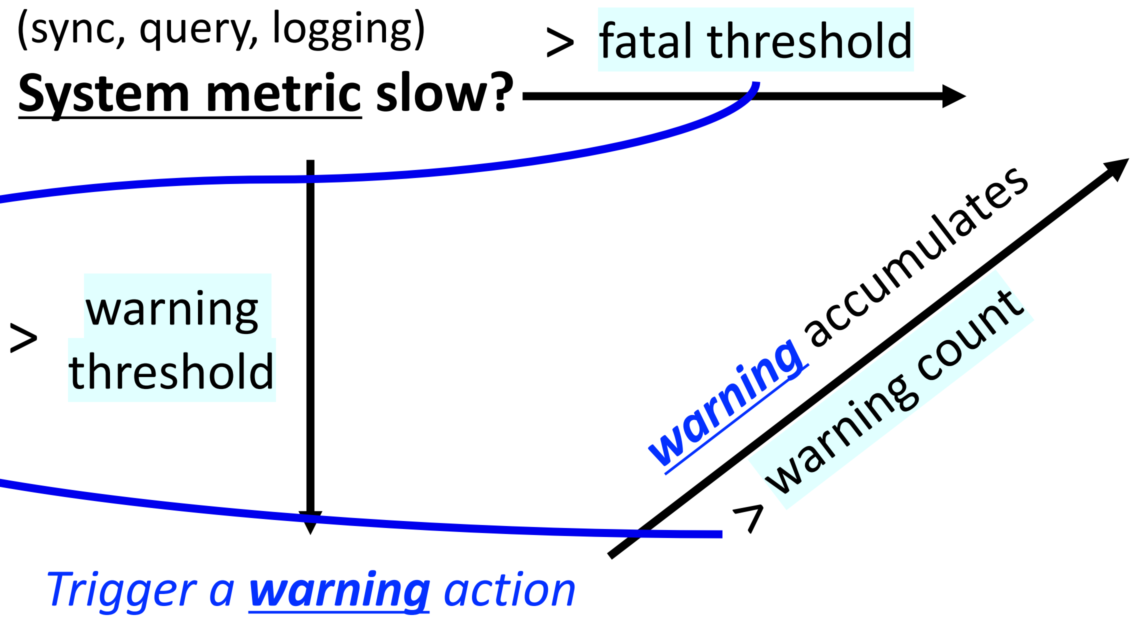
```
public void postSync(syncTime) {
    if (syncTime > 100ms) {
        LOG.INFO(...);
        counter += 1;

    }
}
```



Static-threshold-based slow detection

Slow sync detection in HBase

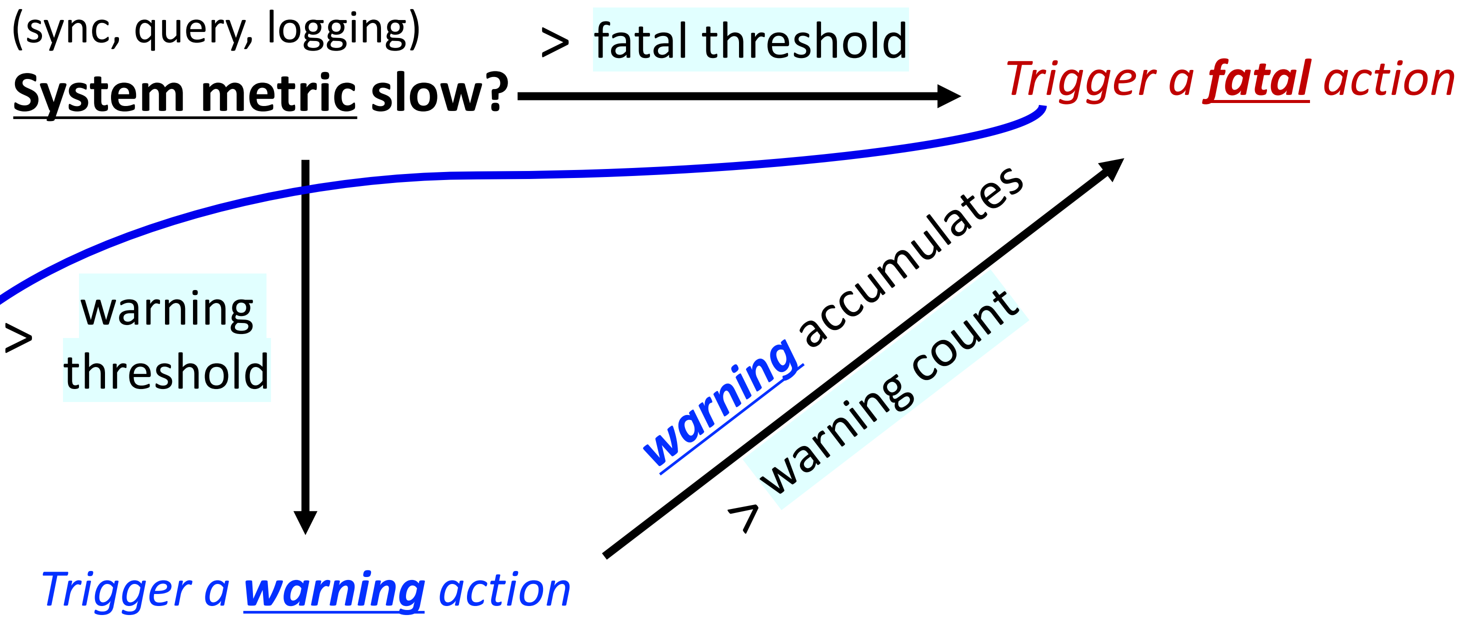
```
public void postSync(syncTime) {
    if (syncTime > 100ms) {
        LOG.INFO(...);
        counter += 1;
        if (syncTime > 10s ||
            counter >= 100) {

        }
    }
}
```



Static-threshold-based slow detection

Slow sync detection in HBase

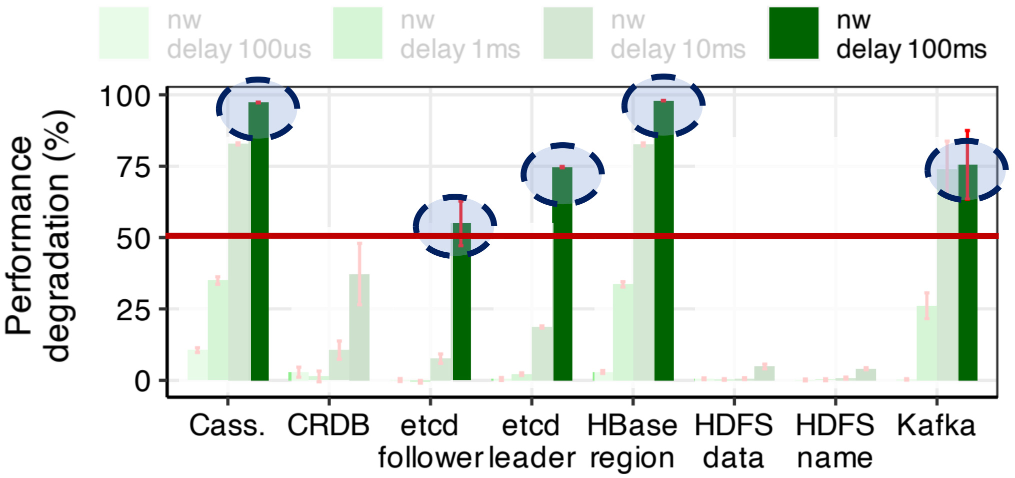
```
public void postSync(syncTime) {
    if (syncTime > 100ms) {
        LOG.INFO(...);
        counter += 1;
        if (syncTime > 10s ||
            counter >= 100) {
            requestLogRoll();
        }
    }
}
```



Developers use static, over-conservative thresholds

System Metric		Static threshold: Warning Threshold	Warning Count	Fatal Threshold
Cassandra	Execution time of last query	500 ms	-	-
CRDB	Execution time of last disk write	5 s	-	20 s
CRDB	Time to flush pending logs	10 s	-	20 s
etcd	`/livez` to check raft loop execution	5 s	3	-
HBase	Time to flush WAL to disk	100 ms	100	10 s
HDFS	Time to get read ACK from datanodes	30 s	-	-
Kafka	Execution time of last request	30 s	-	2 min

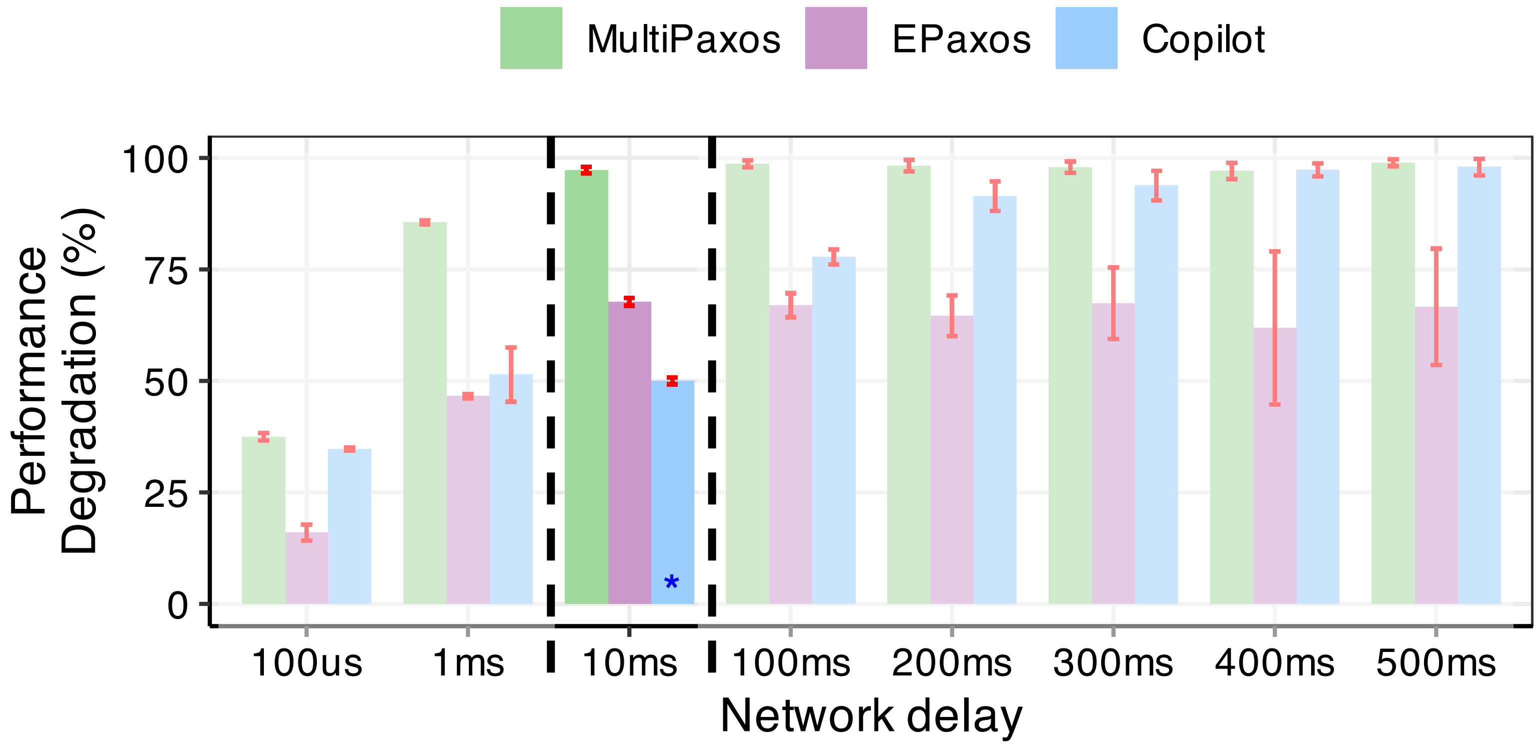
Over-conservative: 100ms, 500ms
5s, 10s, 20s, 30s
2min



$>50\%$ degradation at only 100ms delay!

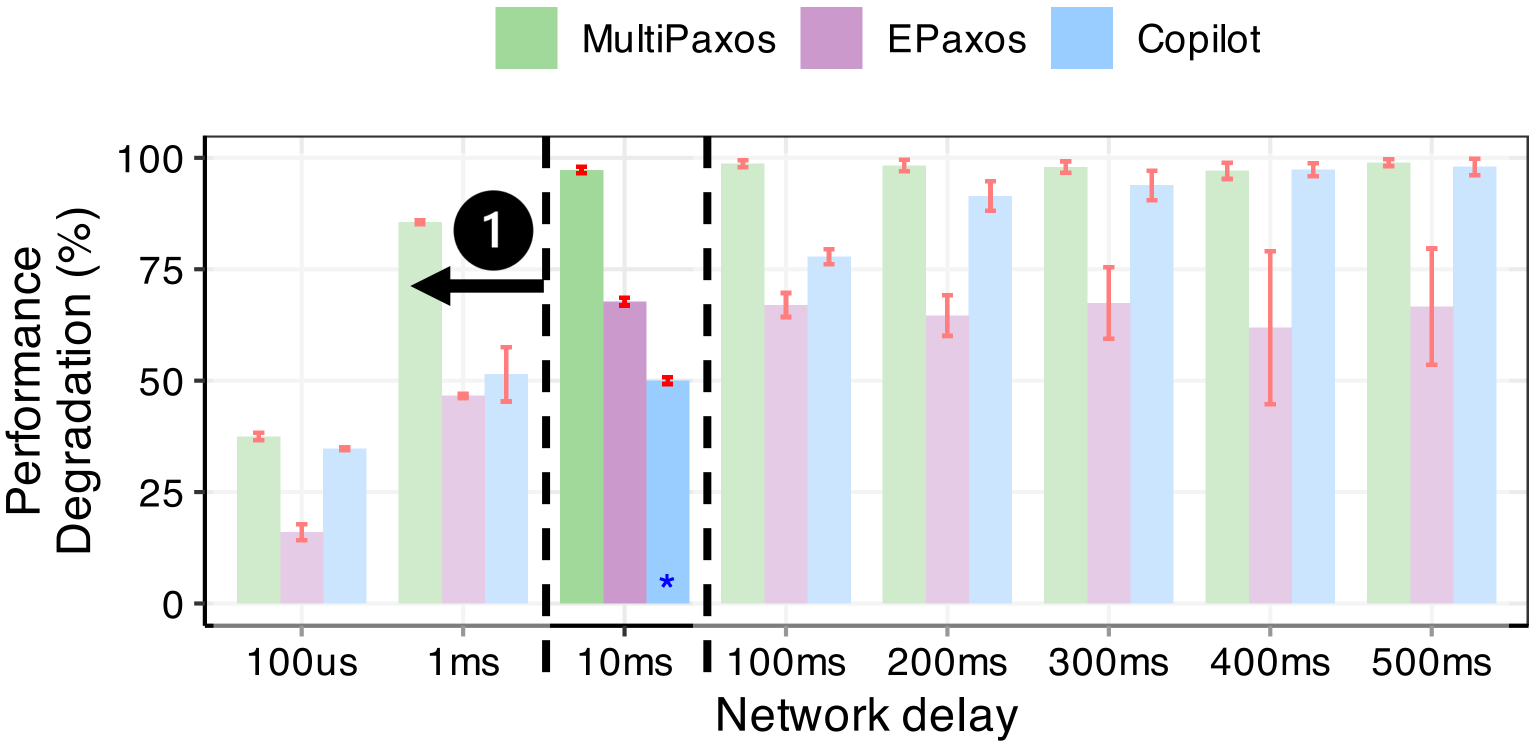
Slow-tolerant protocol suffers from static timeouts

Copilot [OSDI '20]



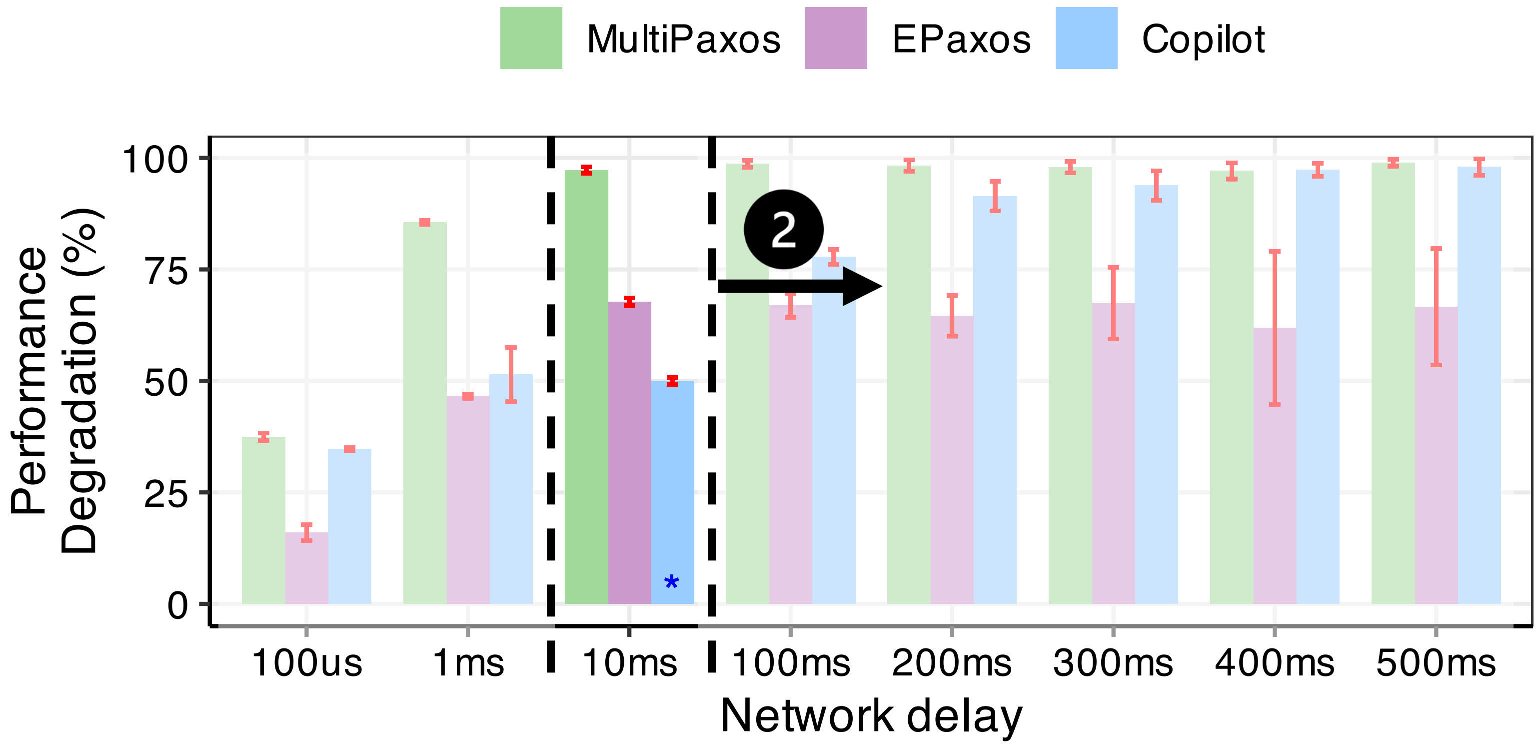
Only optimal **under** ***10ms*** **network delay**

Slow-tolerant protocol suffers from static timeouts



1 Do nothing when delay < ***10ms*** ***(fast-takeover timeout)***

Slow-tolerant protocol suffers from static timeouts



❷ Heartbeat missed at ***100ms***, but still functioning until ***1s***

(BEACON_SENDING_INTERVAL) **(BEACON_MISS_INTERVAL)**

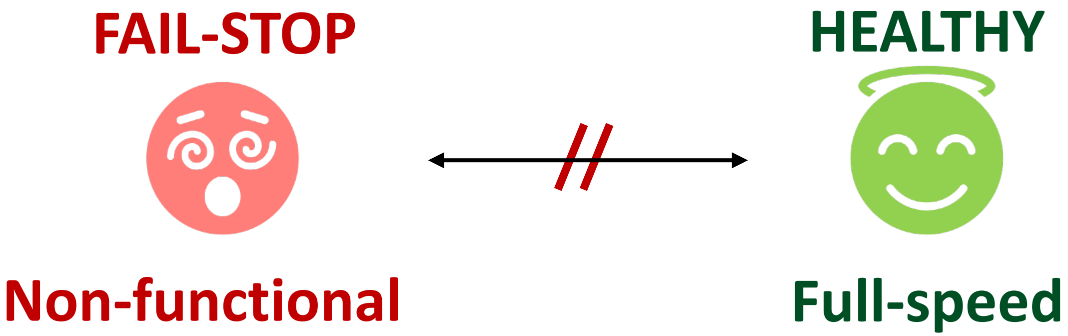
Hard to anticipate real-world slow faults and deployment

Static Threshold

- Cannot change once set ✗→ Dynamic
- Hard to adapt ✗→ Manifest with unforseen workloads, envs, etc.

Slow Fault

Static threhsold works for fail-stop …



Fail-stop has a clear boundary to distinguish

→ **Conservative static thresholds will do!**

```
HDFS.datanode.ConnTimeout = 30s
cassandra.CONNECT_TIMEOUT_MILLIS = 5s
```

… but not for fail-slow!

Fail-slow is ***non-binary*** and ***dynamic***

Hard thresholds won't work well!

Failure detection needs to be ***adaptive***

We propose ADR: Adaptive Detection at Runtime

xxx.java

```
X = ...;
if (X > T){
   ...
}
```

Built-in variable **X**

We propose ADR: Adaptive Detection at Runtime

xxx.java

```
X = ...;
if (X > T){
   ...
}
```

Built-in variable **X**

Value of **X**	30	26	37	69	121	89	21	28
Static threshold **T**	1K	1K	1K	1K	1K	1K	1K	1K

We propose ADR: Adaptive Detection at Runtime

xxx.java

```
X = ...;
if (X > T){
   ...
}
```

Built-in variable `X`

Value of `X`	30	26	37	69	121	89	21	28
Static threshold `T`	1K	1K	1K	1K	1K	1K	1K	1K

*How to build an **adaptive** threshold?*

We propose ADR: Adaptive Detection at Runtime

xxx.java

```
X = ...;
if (X > T){
   ...
}
```

Built-in variable `X`

Value of `X`	30	26	37	69	121	89	21	28
Static threshold `T`	1K	1K	1K	1K	1K	1K	1K	1K

*How to build an **adaptive** threshold?*

Our answer: Use simple statistics of historical values

`99th percentile`

We propose ADR: Adaptive Detection at Runtime

xxx.java

```
X = ...;
if ( X > T ){
   ...
}
```

Built-in variable **X**

Value of **X**	30	26	37	69	121	89	21	28
Static threshold **T**	1K	1K	1K	1K	1K	1K	1K	1K
Adaptive threshold **p99**	54	57	49	48	51	58	53	56

We propose ADR: Adaptive Detection at Runtime

`xxx.java`

```
X = ...;
if (X > T){
   ...
}
```

Built-in variable `X`

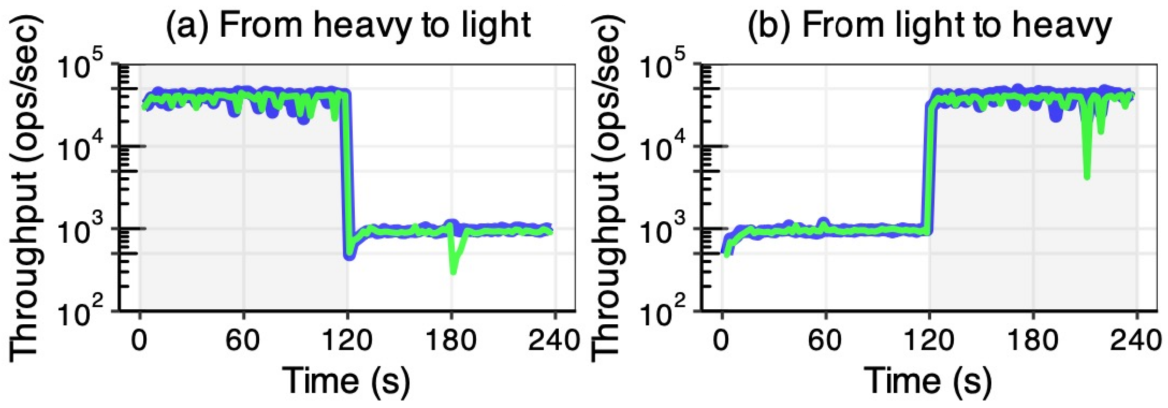
Slow fault?

Value of `X`	30	26	37	69	121	89	21	28
Static threshold `T`	1K	1K	1K	1K	1K	1K	1K	1K
Adaptive threshold `p99`	54	57	49	48	51	58	53	56

Challenge 1: *`p99`* means always *1%* false positives

Challenge 1: *p99* **means always** *1%* **false positives**

Challenge 2: ***Real slow faults*** **or** ***normal workload variations*****?**



Workload intensity may affect system states

We observe:

Workload variations **can be well described by the** ***update frequency*** **of variables**

xxx.java

```
X = ...;
if (X > T){
  ...
}
```

Built-in variable

The number of times **X** gets updated in a second

Case 1: Heavier workloads

❶ **Value**	30	26	37	69	121	89	21	28
❷ **Update Frequency** resp./second	101	110	105	210	240	220	101	104

Frequency ↑ = Workload ↑

Normal variation!

xxx.java

```
X = ...;
if (X > T){
   ...
}
```

Built-in variable **X**

```
xxx.java
X = ...;
if (X > T){
   ...
}
```

Built-in variable **X**

Case 1: Heavier workloads

① **Value**	30	26	37	69	121	89	21	28
② **Update Frequency** resp./second	101	110	105	210	240	220	101	104

Frequency ↑ = Workload ↑

Normal variation!

Case 2: Lighter workloads

❶ **Value**	30	26	37	26	29	31	21	28
❷ **Update Frequency** resp./second	101	110	105	46	51	37	101	104

Frequency ↓ + Value — = Workload ↓

Normal variation!

xxx.java

```
X = ...;
if ( X > T ){
   ...
}
```

Built-in variable **X**

Case 1: Heavier workloads

1 Value	30	26	37	69	121	89	21	28
2 Update Frequency resp./second	101	110	105	210	240	220	101	104

Frequency ↑ = Workload ↑

Normal variation!

Case 2: Lighter workloads

1 Value	30	26	37	26	29	31	21	28
2 Update Frequency resp./second	101	110	105	46	51	37	101	104

Frequency ↓ + Value — = Workload ↓

Normal variation!

Case 3: Slow faults

1 **Value**	30	26	37	69	121	89	21	28
2 **Update Frequency** resp./second	101	110	105	25	31	22	101	104

Frequency ↓ + Value ∼ = 😈

Slow Faults!

ADR as a plug-in: Replacing existing static logic

Slow sync detection in HBase

```
public void postSync(syncTime) {
    if (syncTime > 100ms) {
        LOG.INFO(...);
        if (syncTime > 10s) {
            requestLogRoll();
        }
    }
}
```

Warning threshold

Fatal threshold

ADR as a plug-in: Replacing existing static logic

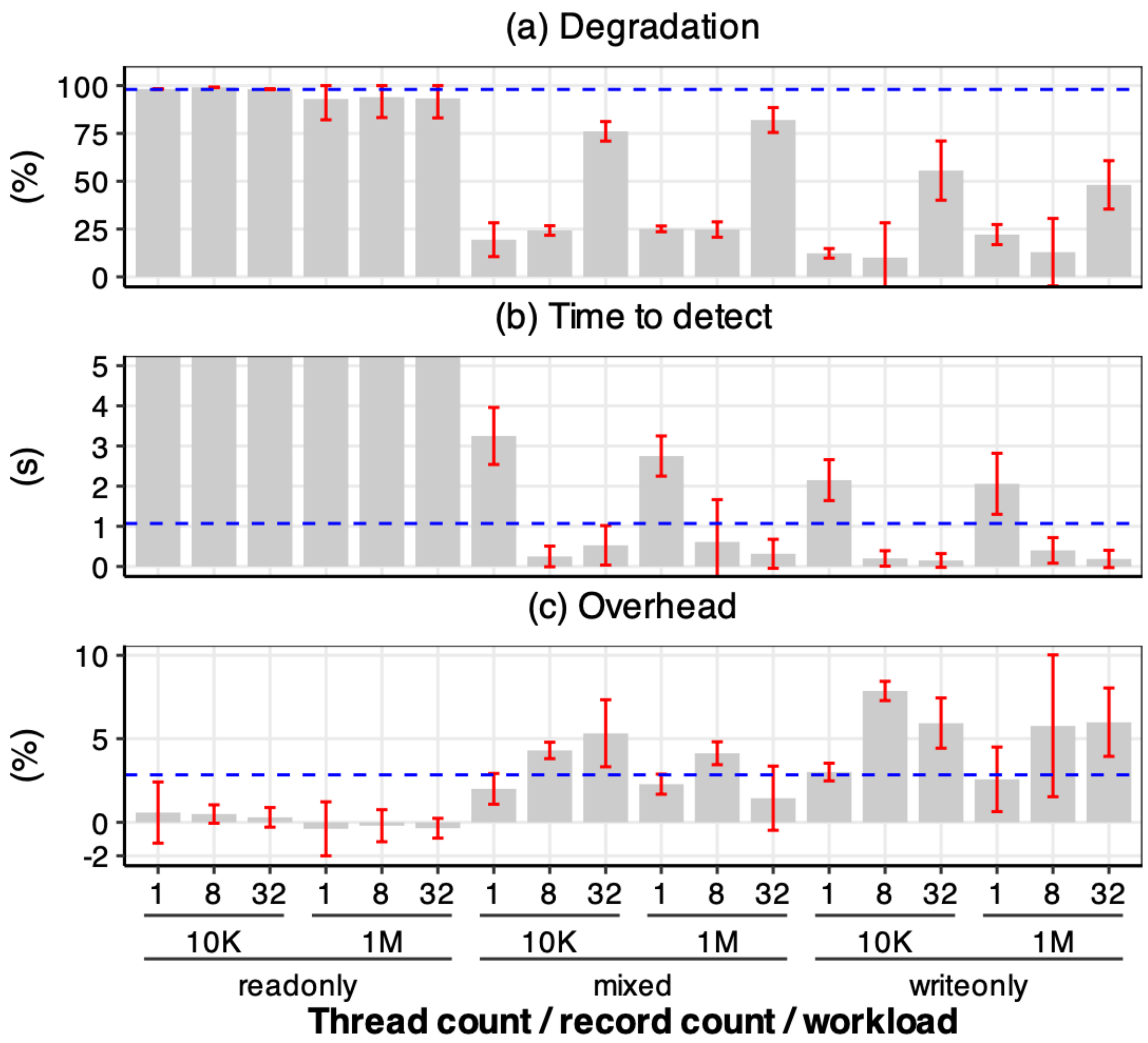
Slow sync detection in HBase

```
public void postSync(syncTime) {
    if (syncTime > 100ms) {
        LOG.INFO(...);
        if (syncTime > 10s) {
            requestLogRoll();
        }
    }
}
```

Slow sync detection using ADR

```
public void postSync(syncTime) {
    if (ADR.isWarn(syncTime, '>', 100ms)) {
        LOG.INFO(...);
        if (ADR.isFatal(syncTime, '>', 10s)) {
            requestLogRoll();
        }
    }
}
```

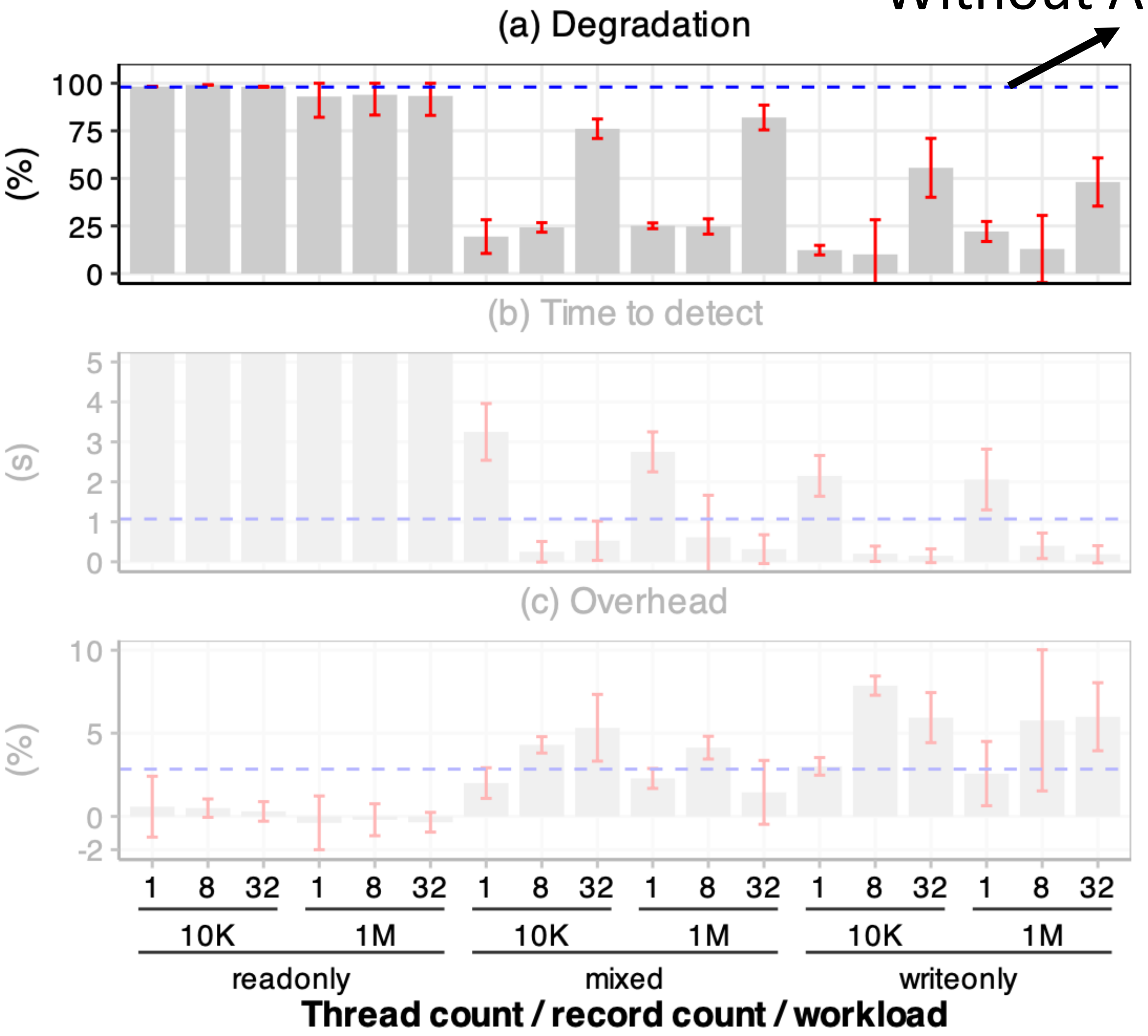
Evaluation



(a) Degradation
100
75
50
25
0
(%)
(b) Time to detect
5
4
3
2
1
0
(s)
(c) Overhead
10
5
0
-2
(%)
1 8 32 1 8 32 1 8 32 1 8 32 1 8 32 1 8 32
10K 1M 10K 1M 10K 1M
readonly mixed writeonly
Thread count / record count / workload

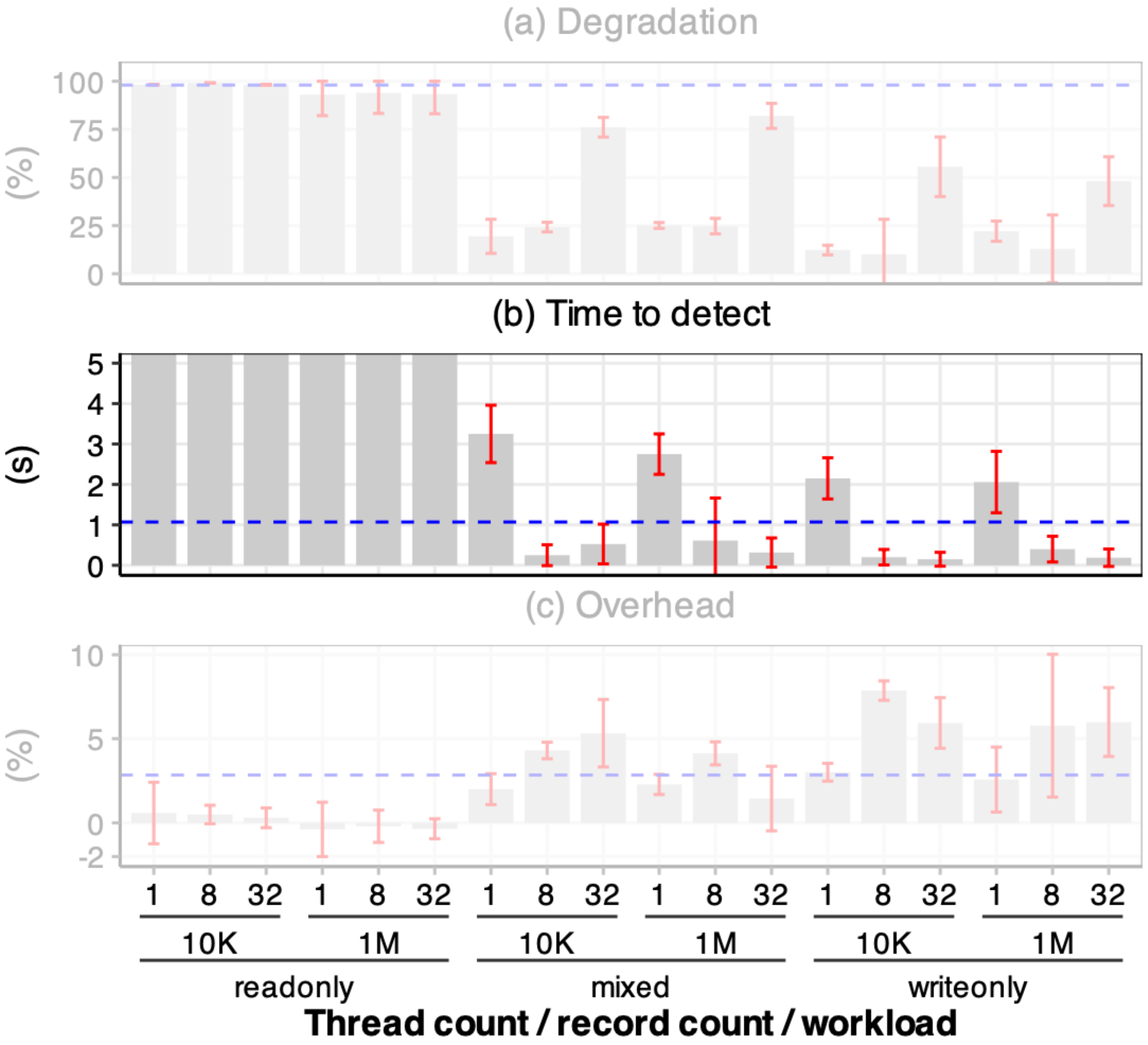
Evaluation

Without ADR: **97%** degradation



Reduce degradation by 16-90%

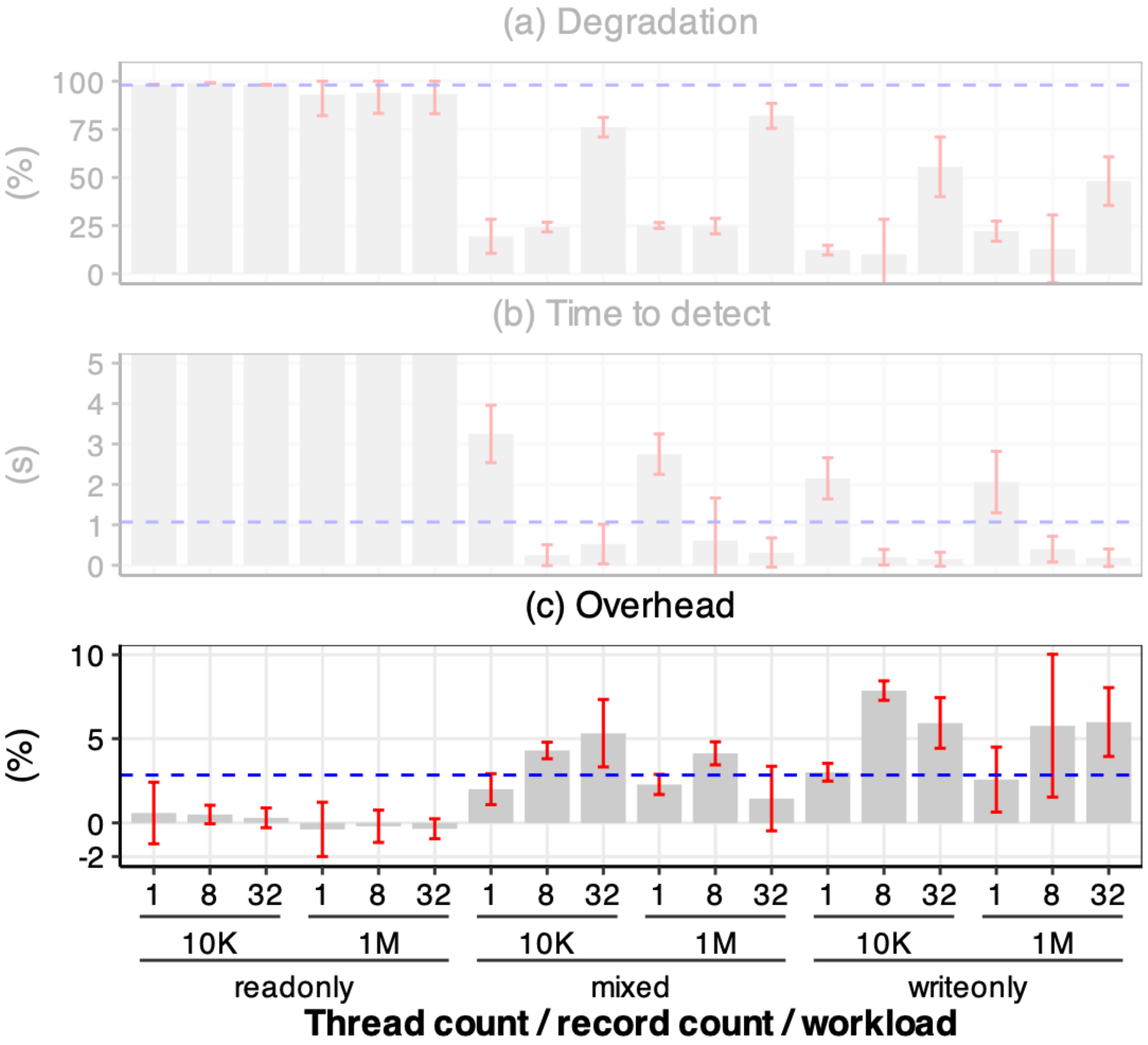
Evaluation



Reduce degradation by 16-90%

Timely detection in seconds

Evaluation



Reduce degradation by 16-90%

Timely detection in seconds

Minimal 2.8% average overhead

Conclusion

1. **Automated testing pipeline to measure slow-fault tolerance**
2. **Slow-fault tolerance is nuanced and sensitive to**
   - *Slow faults: Severity, type, location, duration, start time*
   - *Deployment: Resources, configs, workloads*
3. **Detecting slowness with static thresholds is insufficient**
4. **ADR – lightweight, adaptive slow-fault detection library at runtime**

The testing pipeline and ADR are available at
https://github.com/OrderLab/xinda