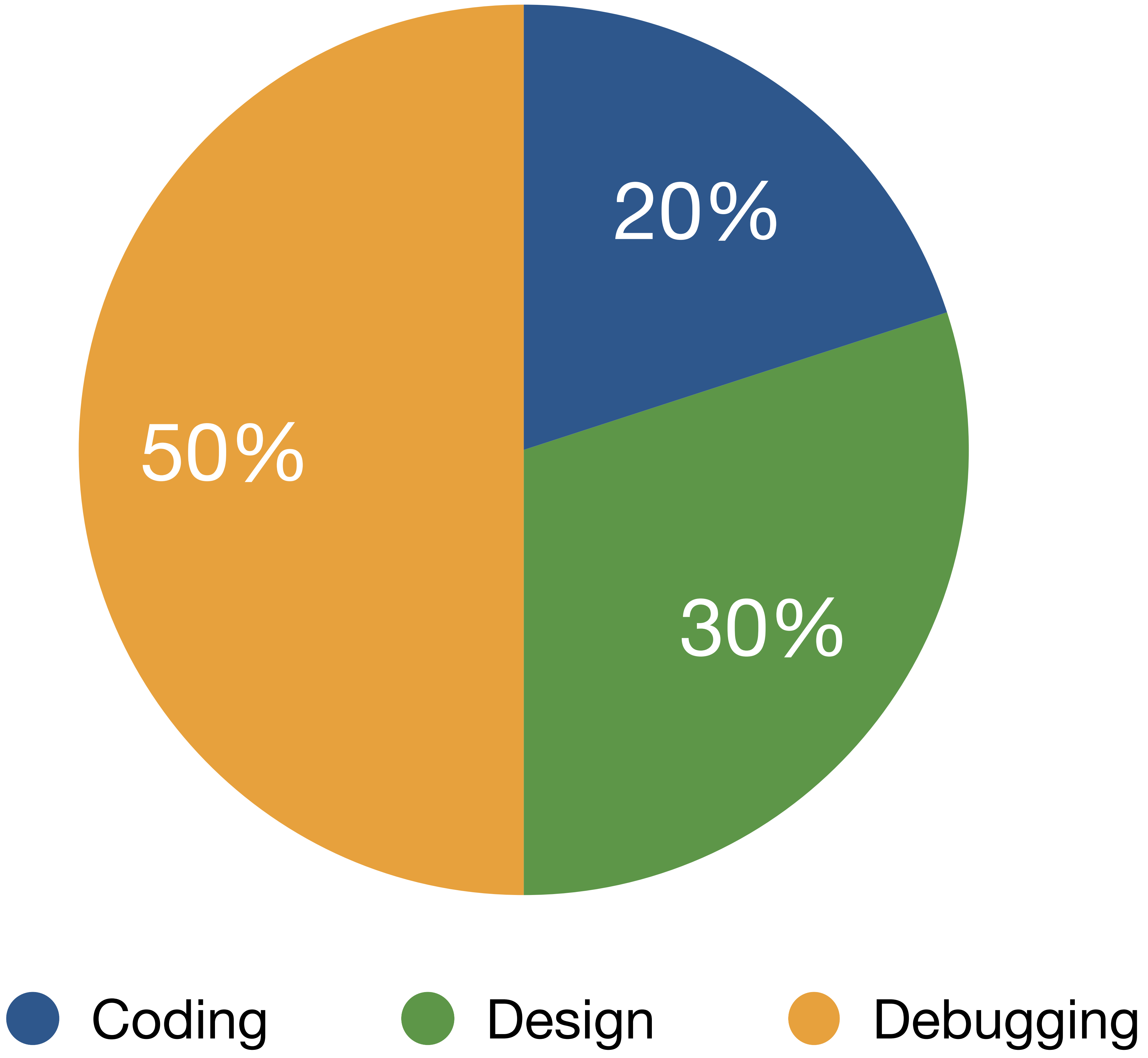
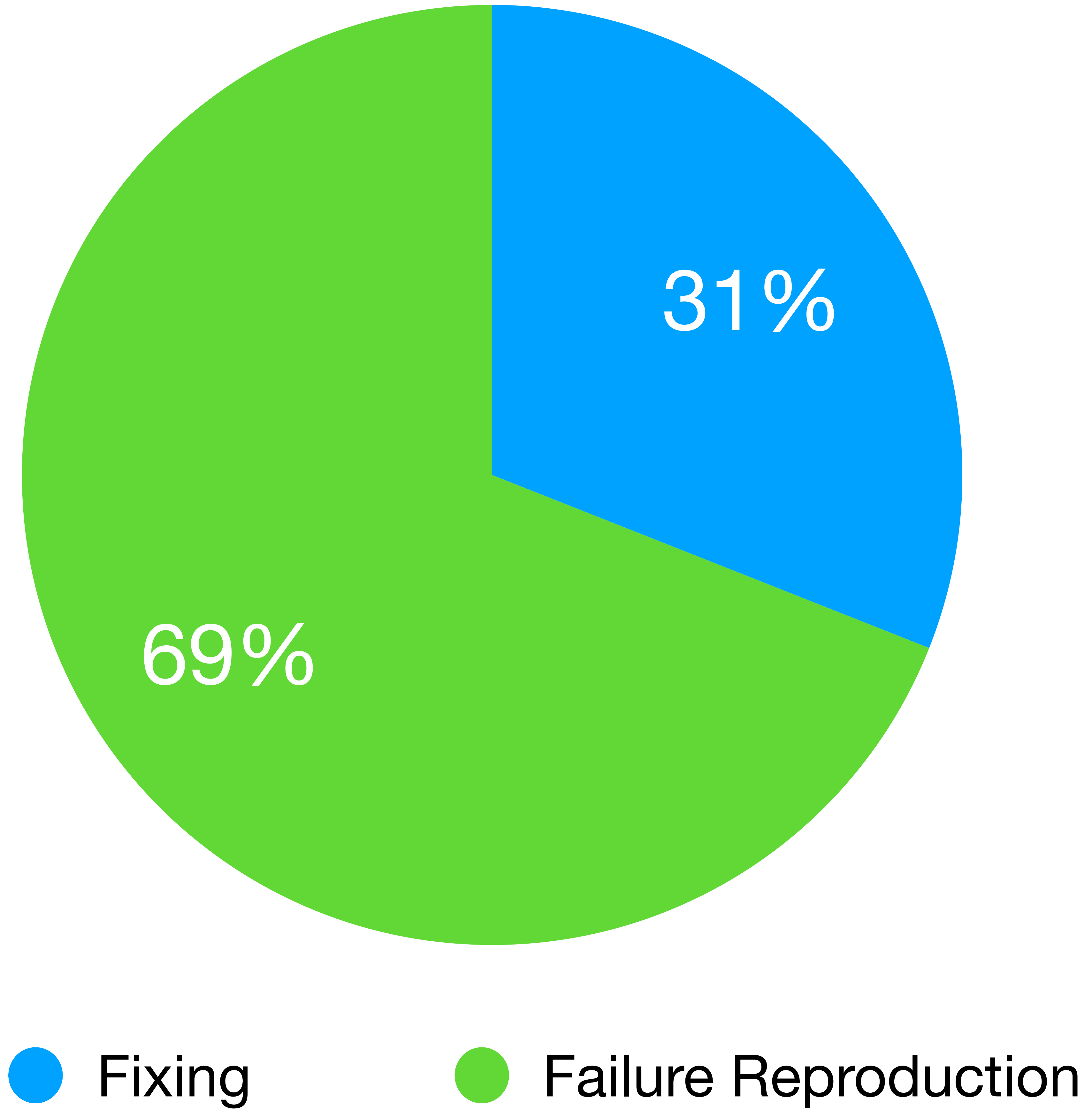
Efficient Reproduction of Fault-Induced Failures in Distributed Systems with Feedback-Driven Fault Injection

Jia Pan, Haoze Wu, Tanakorn Leesatapornwongsa, Suman Nath, Peng Huang

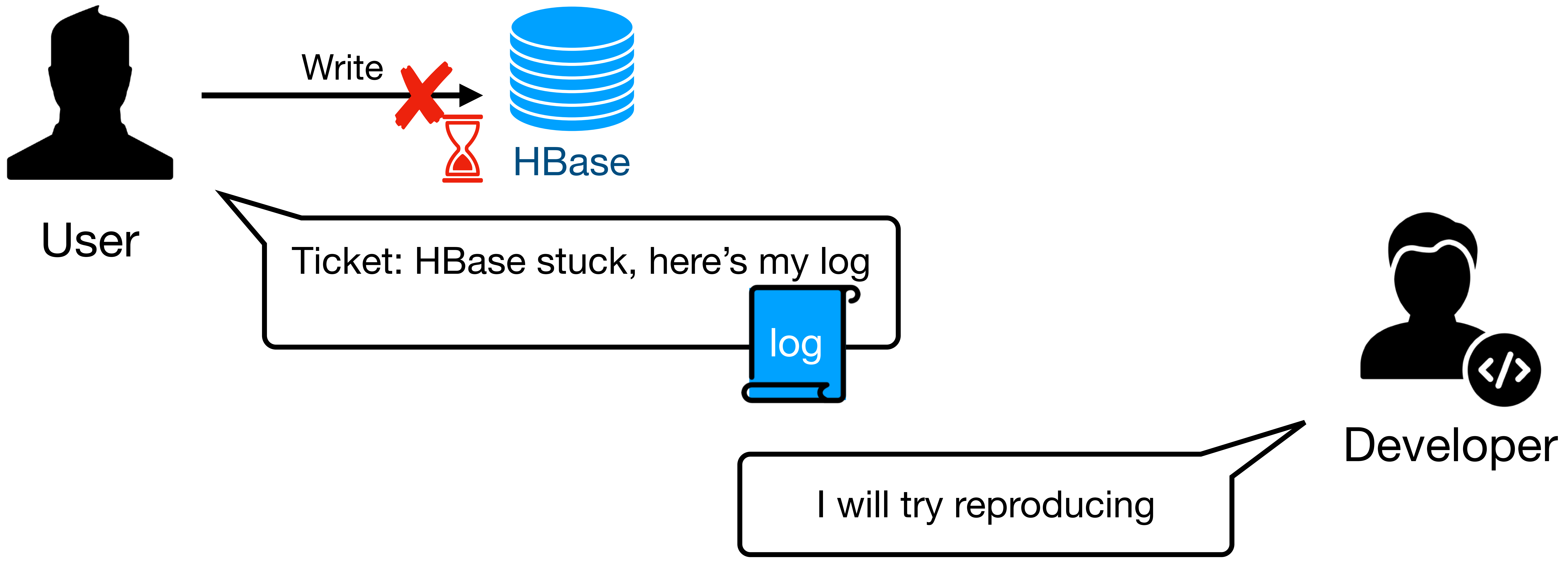
Reproducing Distributed System Failure Is Hard





Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach, Zhang et al., 2017

A Real-World Failure: HBase-25905



A Real-World Failure: HBase-25905

[ERROR] WAL Append Timeout

OK. Try kill during WAL Append
But cannot reproduce…

Root cause not readily available

A Real-World Failure: HBase-25905

During the next 19 days

log

[Info]: Exception X in DISK I/O tolerated ×N times
[Info]: Exception Y in Network I/O tolerated ×M times
...

Developer

Any of those exception could be root cause

Knowing which external fault is necessary

A Real-World Failure: HBase-25905

During the next 19 days

Exception in `channelRead0()` seems interesting

Developer

HBase

`hdfs_write()`

`hdfs_read()` → `channelRead0()` *throw* ⟹ Log: WAL Append Timeout

`...()`

?

Many callers
Many invocations

But which invocation?

Timing is important

A Real-World Failure: HBase-25905

19 days later

HBase

`hdfs_write()` One specific time

`channelRead0()` Disconnected temporarily
throw

⟹ WAL roll over **Edge case bug**

⟹ Log: WAL Append Timeout Symptom

Developer

Finally found & fixed

Existing Work

Reproduce the input (external API) for a given failure

- Not suitable for fault-induced failures

Fault injection testing and chaos engineering

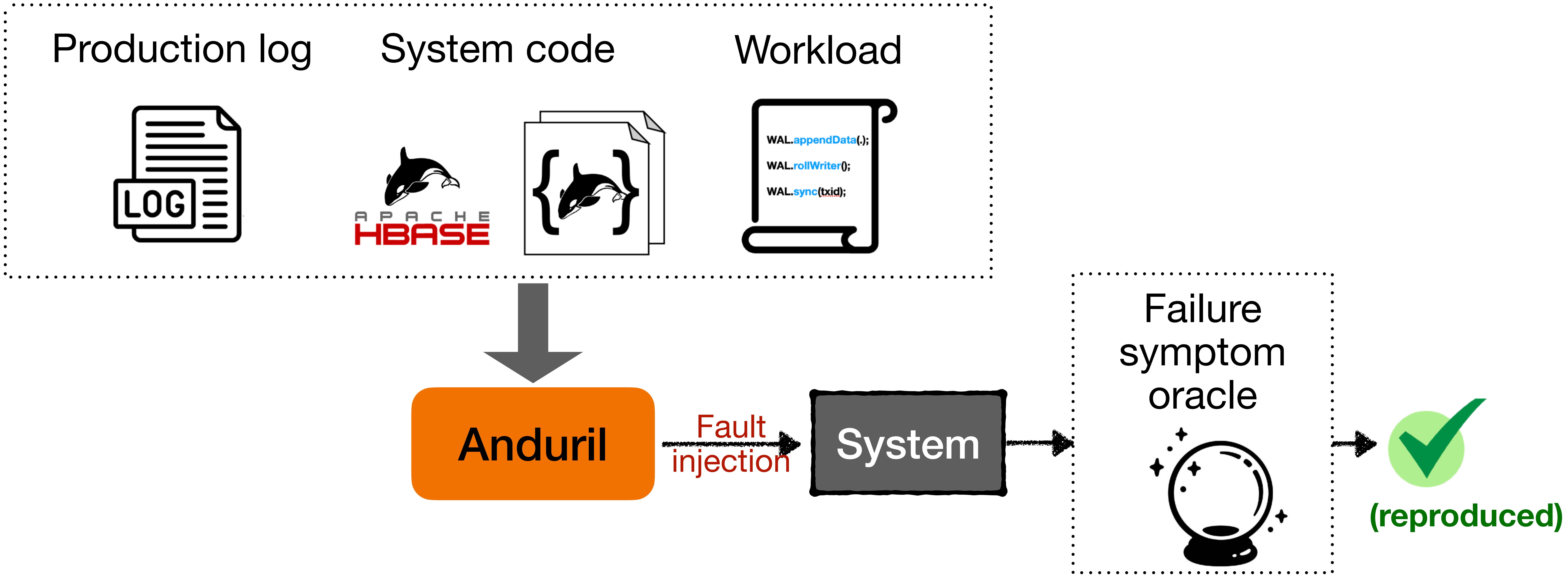
- Only for bug finding

Record and replay

- Intrusive and high runtime overhead

Our Goal

Reproduce a **specific** ***fault-induced failure*** **quickly without** runtime overhead



Challenges

Exploding injection space

- 18 K – 28 K static fault sites (program points in system code)
- 1 K+ occurrences in some fault site
- Need to find the root-cause fault site and proper timing

Faults buried inside logs

- Tolerated faults introduce many **noises** in the production log
- An error message may **miss** key stack traces

Inefficient to enumerate all possibilities

Key Ideas of Anduril

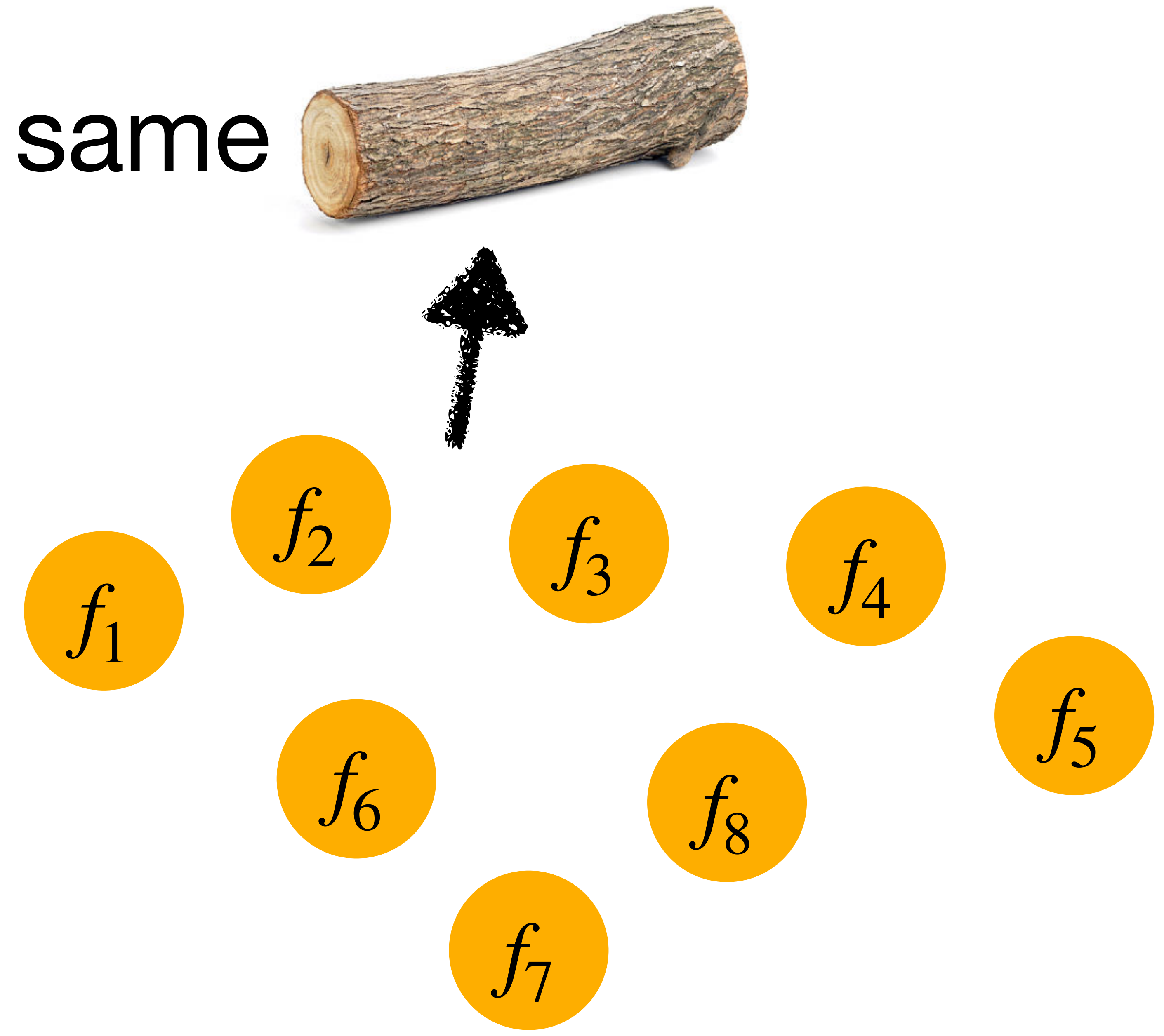
Prune fault sites that are irrelevant to a given failure

- Static causal graph consisting of program points potentially related to the failure symptom

Iteratively search in the injection space through feedback

- Multi-round feedback algorithm dynamically adjusting the priorities of the fault candidates to try next

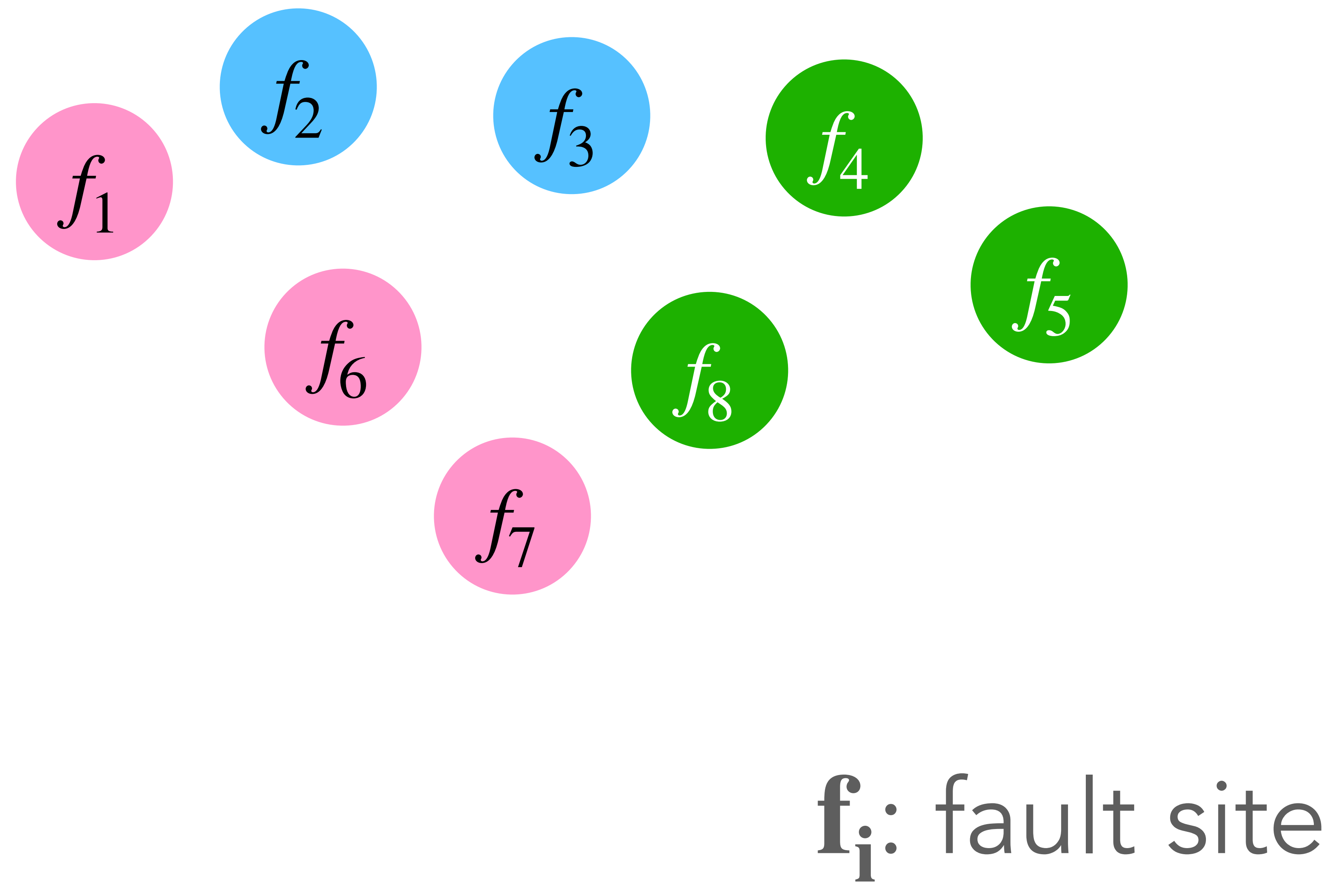
Deprioritize Similar Faults If One Injection Fails



f_i: fault site

- **Insight:** Many faults have **same effect**
  - **Fault traits**

Deprioritize Similar Faults If One Injection Fails

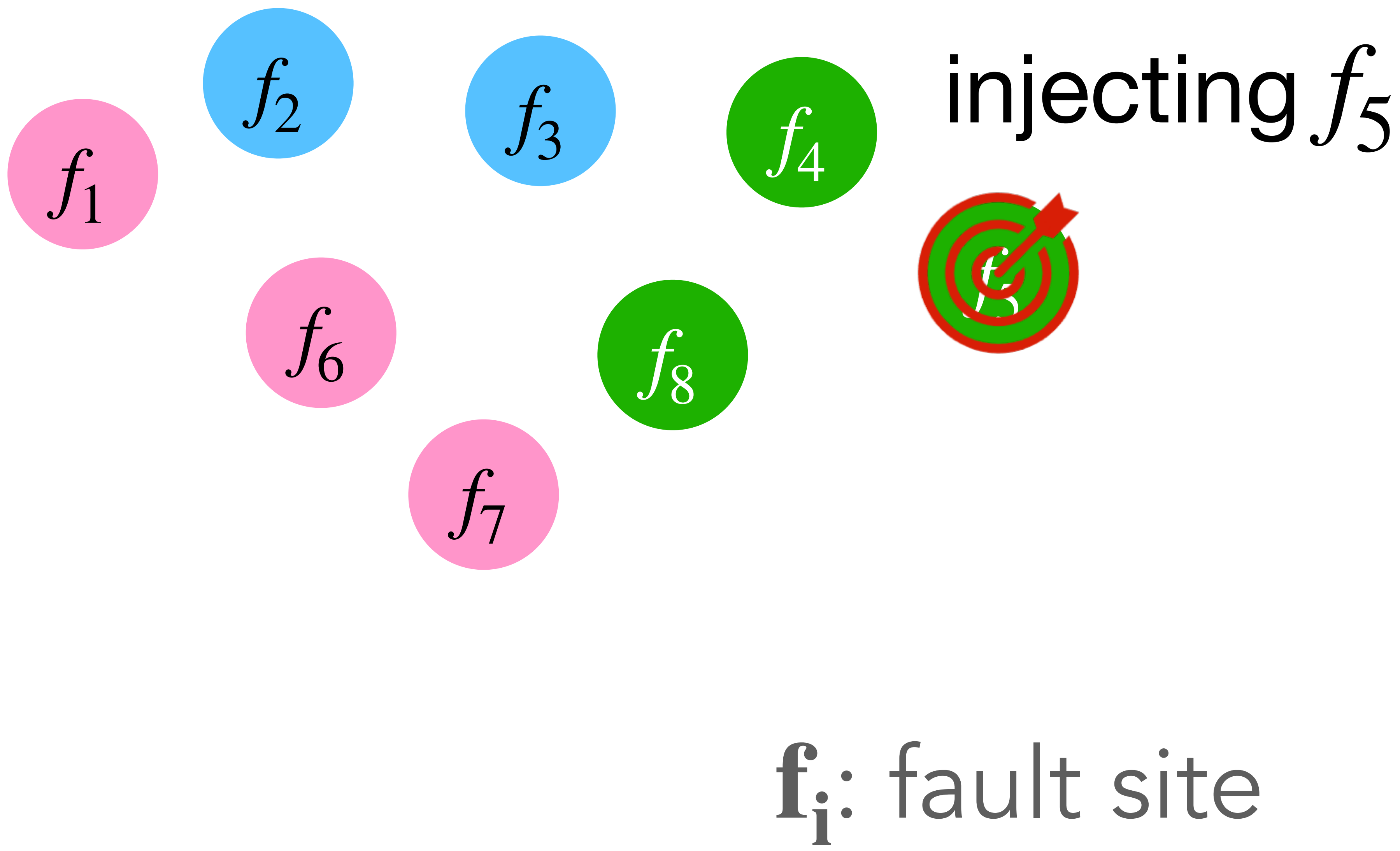


Trait-based Fault-site Grouping

- **Insight:** Many faults have **same effect**
  — **Fault traits**

- Prioritize faults based on their **traits**

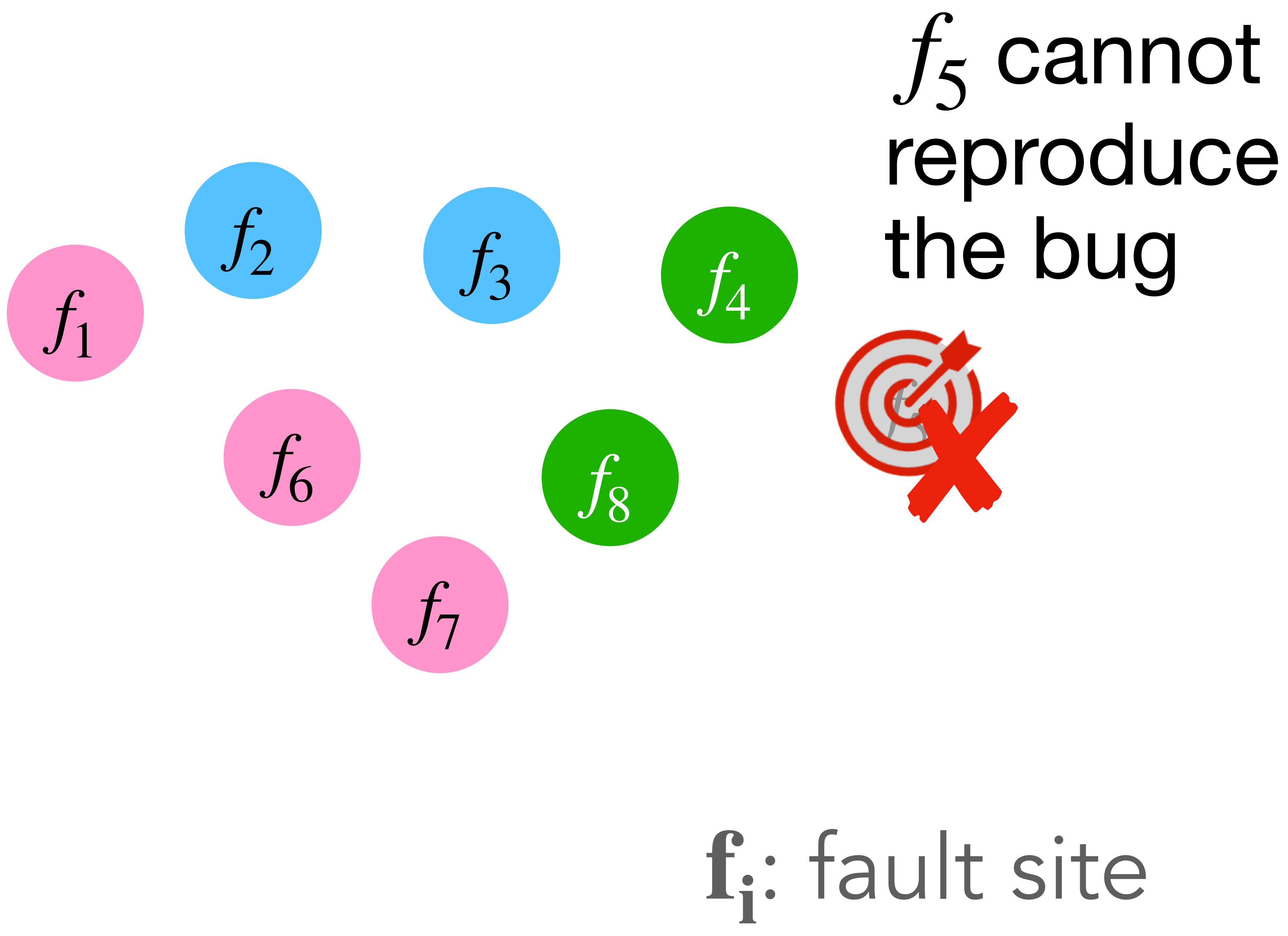
Deprioritize Similar Faults If One Injection Fails



Trait-based Fault-site Grouping

- **Insight:** Many faults have **same effect**
  - **Fault traits**
- Prioritize faults based on their **traits**

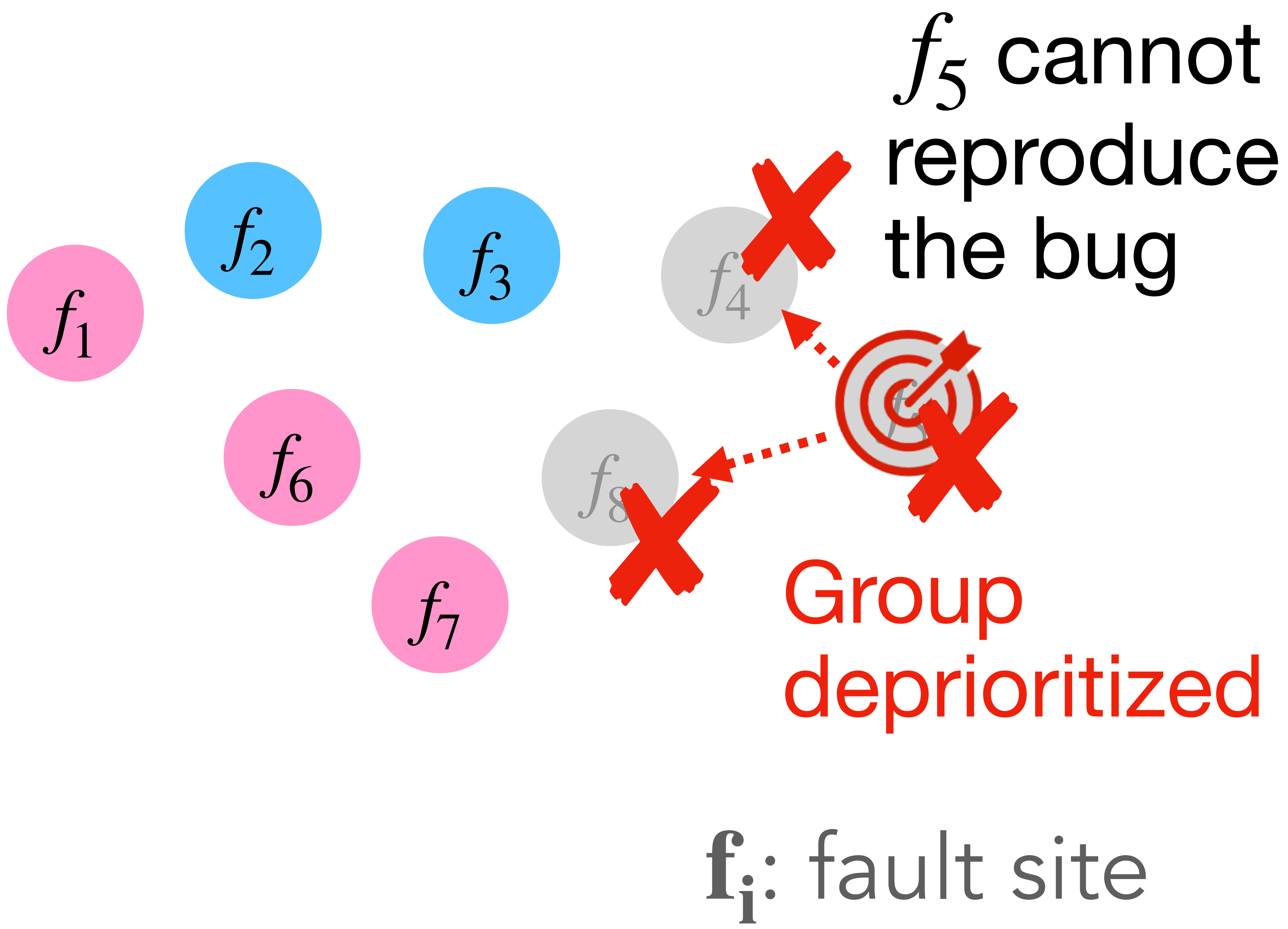
Deprioritize Similar Faults If One Injection Fails



Trait-based Fault-site Grouping

- **Insight:** Many faults have **same effect**
- Prioritize faults based on their **traits**
- If injecting one fault did not trigger the failure, other faults sharing the same traits also likely won't reproduce it

Deprioritize Similar Faults If One Injection Fails



Trait-based Fault-site Grouping

- **Insight:** Many faults have **same effect**
- Prioritize faults based on their **traits**
- If injecting one fault did not trigger the failure, other faults sharing the same traits also likely won't reproduce it

Define Fault Traits

Complete execution state (PC, stack trace, memory)?

- Too expensive to track
- Severely slow down failure reproduction

Triggered try-catch blocks?

- Still require intrusive online recording

```
try {
  …
} catch (…) {
  // handling logic
}
```

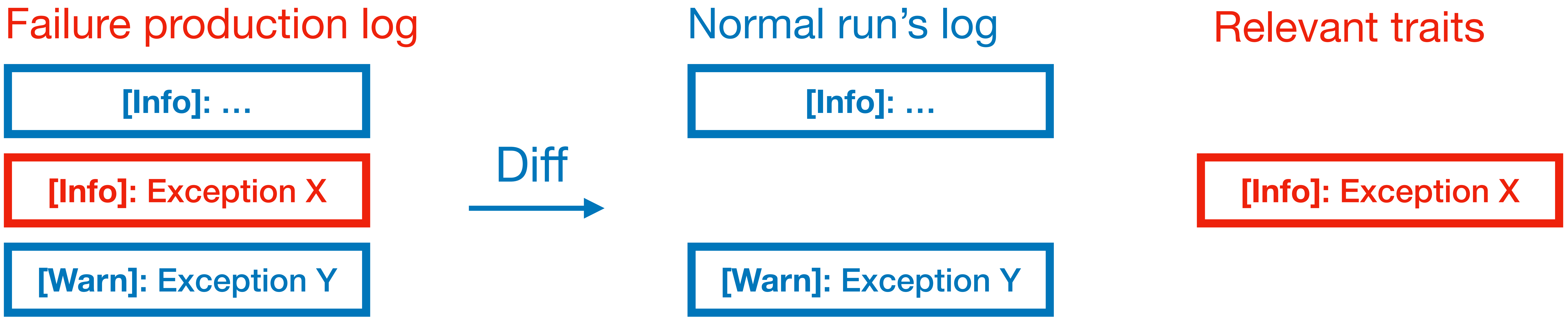
Define Fault Traits

Use log messages **to approximate fault traits**

- No additional runtime overhead
- Enable static estimation of the ***unexplored faults***' traits
- Abstract high-level system state from low-level execution details

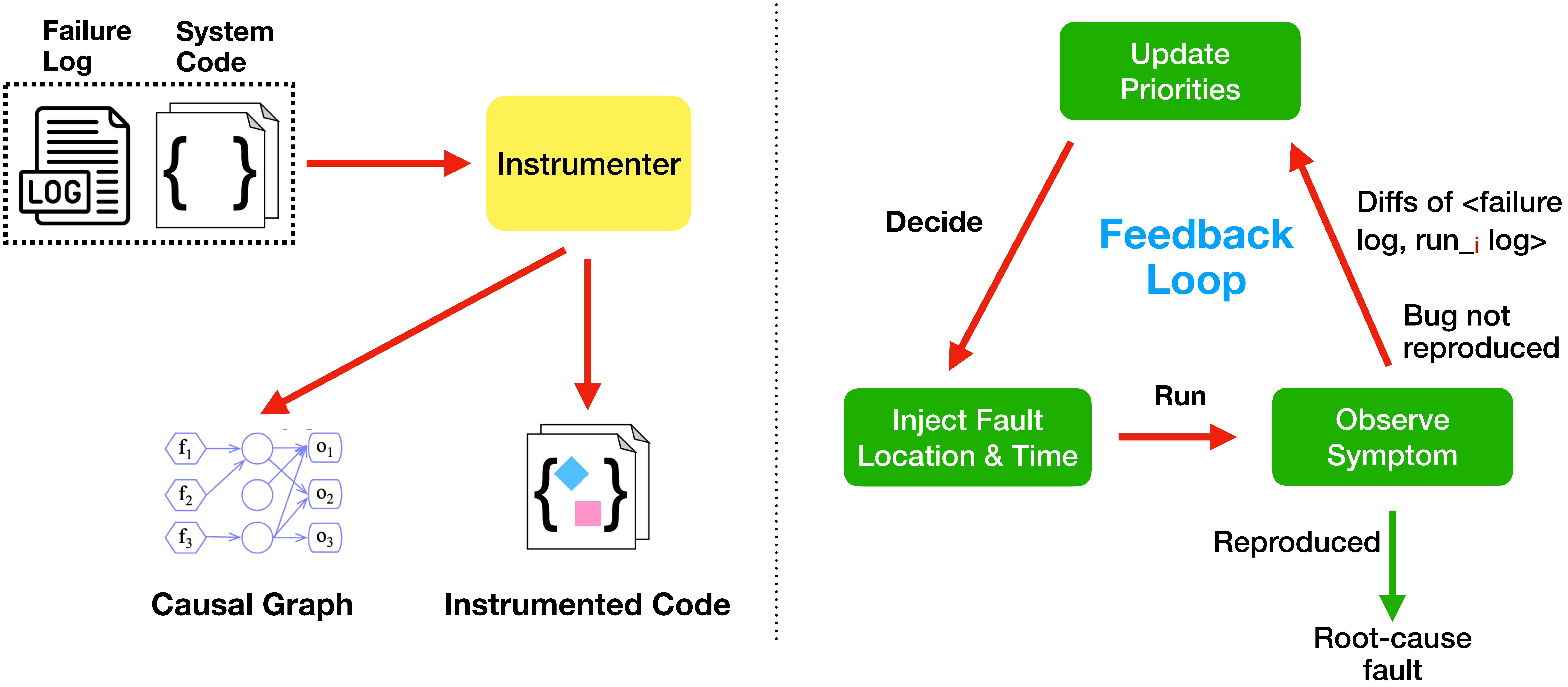
Identifying Relevant Traits

Some log messages not relevant to the failure



- Standard diff does not work:
  - **Concurrency:** log messages interleave across runs
  - Timestamp makes each log message appears unique
- Solution: Sanitize and partition logs by thread before diff

Workflow of Anduril



Static Causal Graph

Objective:

- Identify which faults may cause a fault trait

Computed recursively

- Use “jump” strategy proposed in Pensive for scalability
- May introduce false edges
- Rely on dynamic feedback to address them

```
channelRead(x) {
 socket.read();
}


try {
 channelRead(…)
} catch(…) {
  log(“bad”)
}
```

e_4 External exception

e_1 Trait

Static Causal Graph

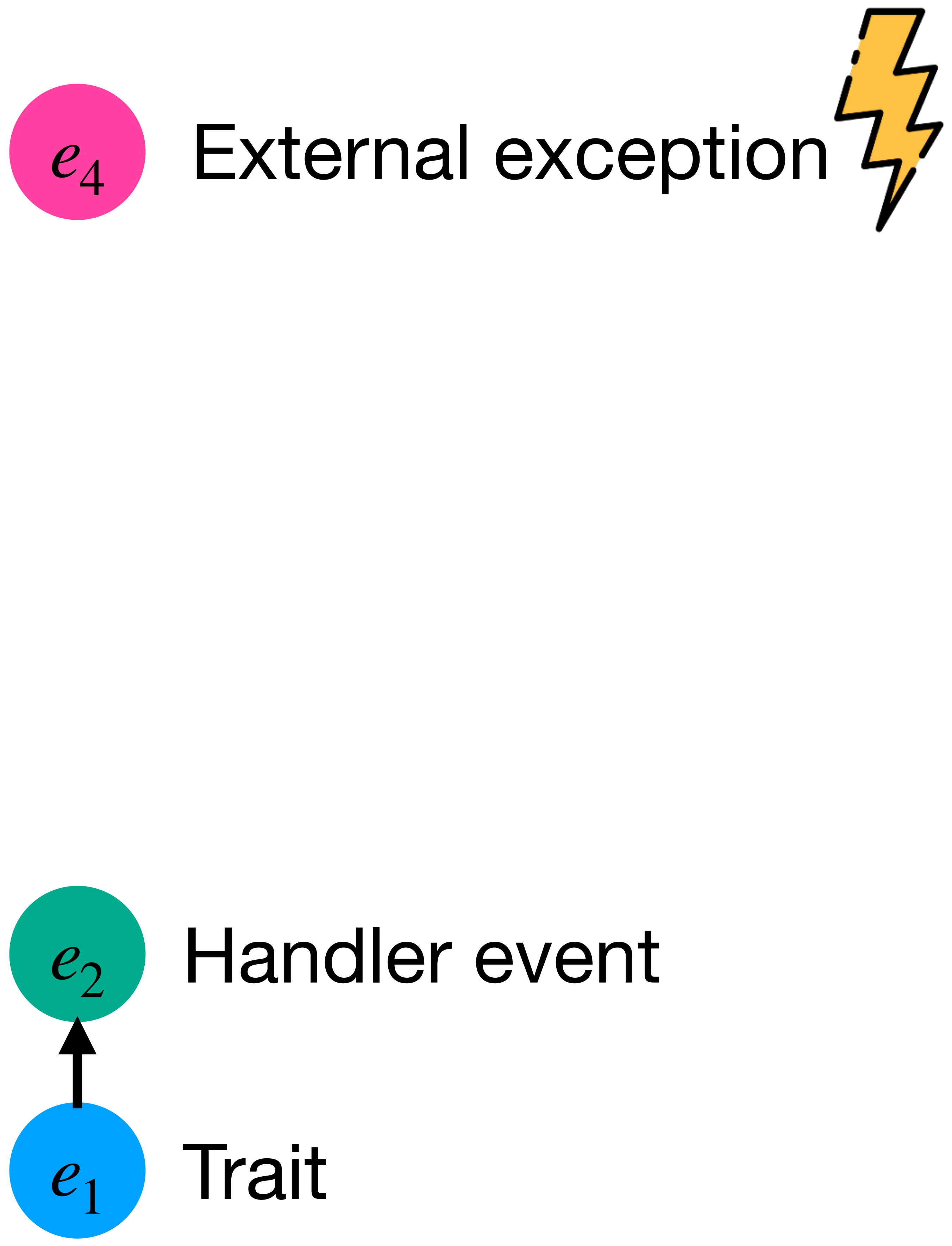
Objective:

- Identify which faults may cause a fault trait

Computed recursively

- Use “jump” strategy proposed in Pensive for scalability
- May introduce false edges
- Rely on dynamic feedback to address them

```
channelRead(x) {
 socket.read();
}


try {
 channelRead(…)
} catch(…) {
  log(“bad”)
}
```



Static Causal Graph

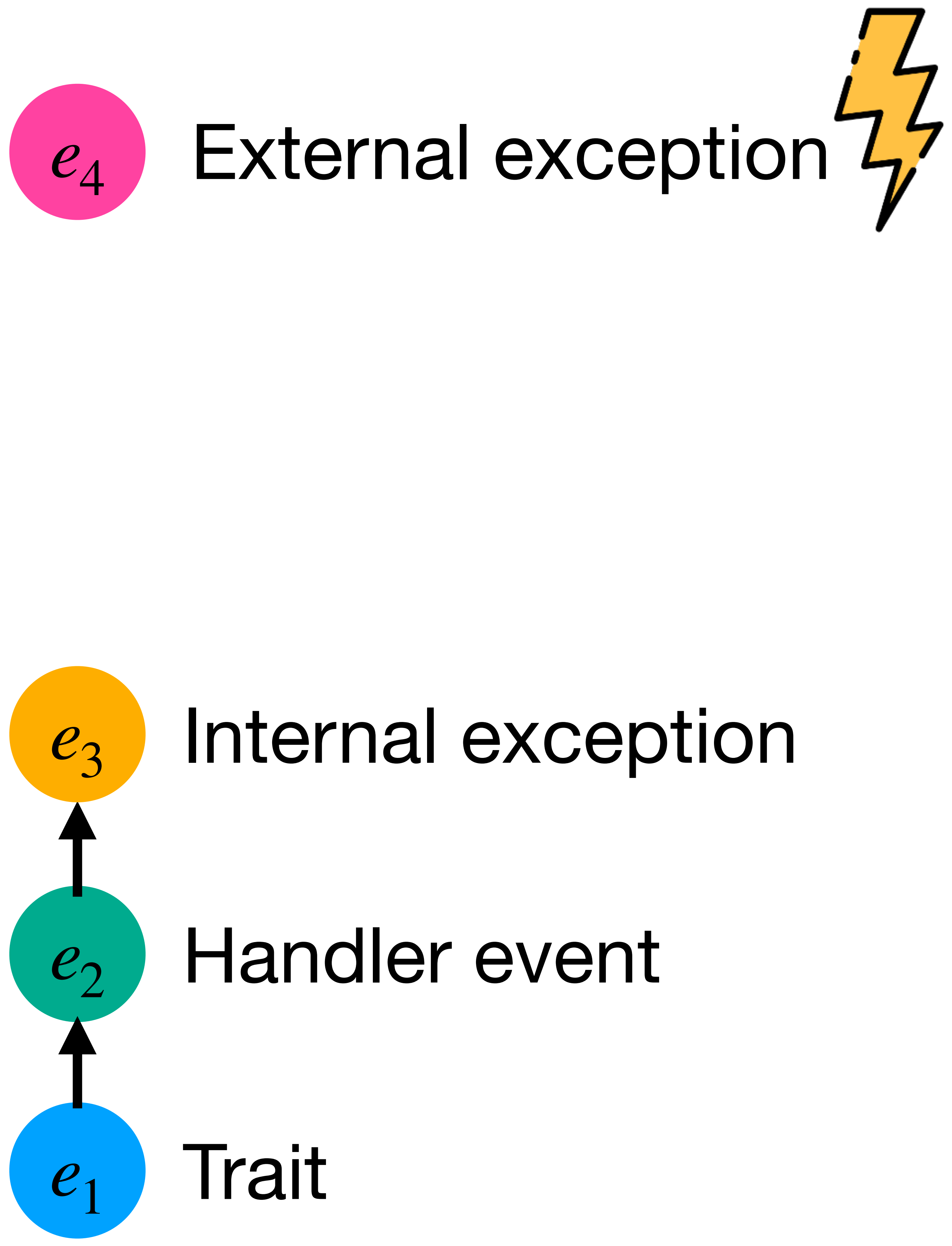
Objective:

- Identify which faults may cause a fault trait

Computed recursively

- Use “jump” strategy proposed in Pensive for scalability
- May introduce false edges
- Rely on dynamic feedback to address them

```
channelRead(x) {
  socket.read();
}


try {
  channelRead(…)
} catch(…) {
  log(“bad”)
}
```



Static Causal Graph

Check the paper for details!

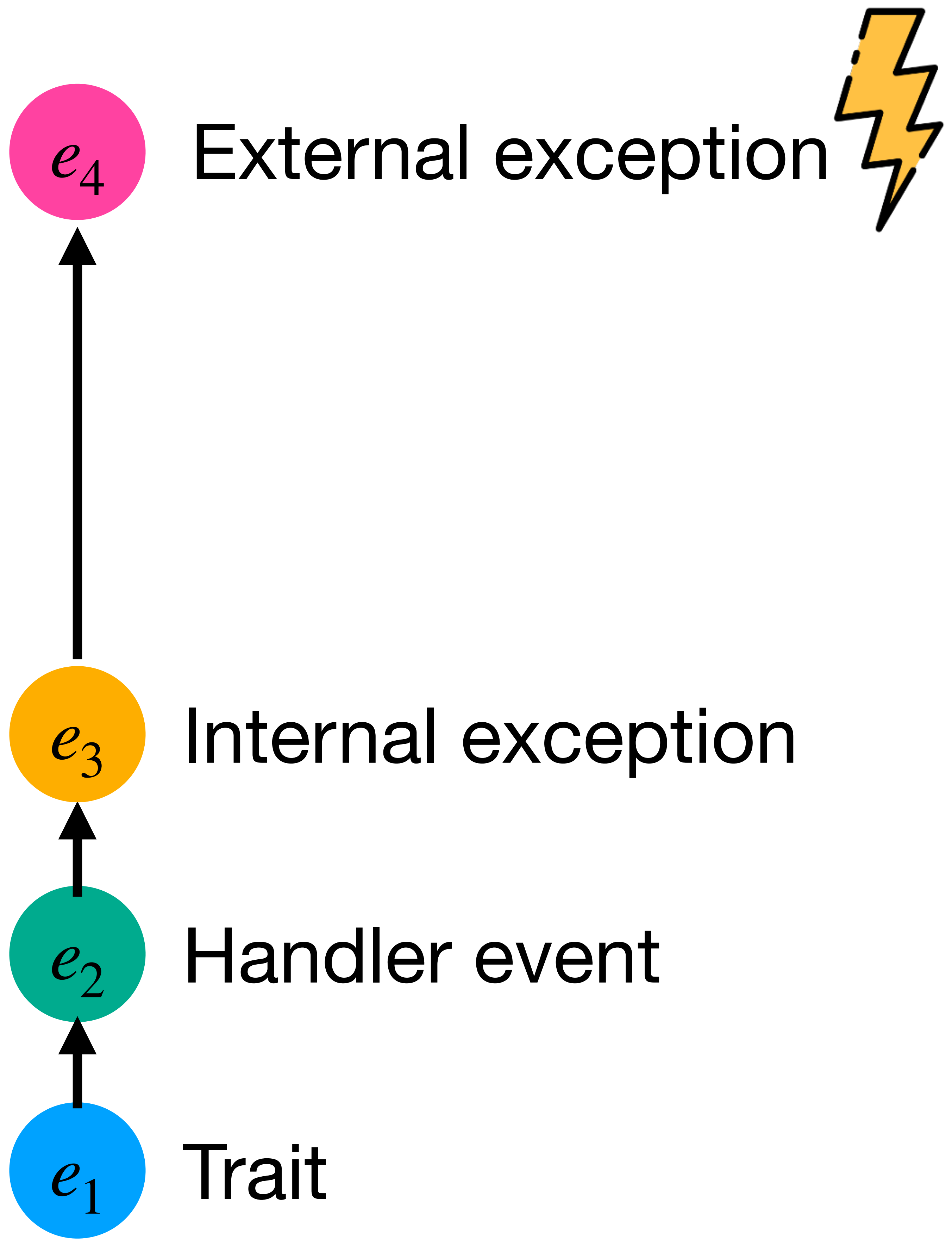
Objective:

- Identify which faults may cause a fault trait

Computed recursively

- Use “jump” strategy proposed in Pensive for scalability
- May introduce false edges
- Rely on dynamic feedback to address them

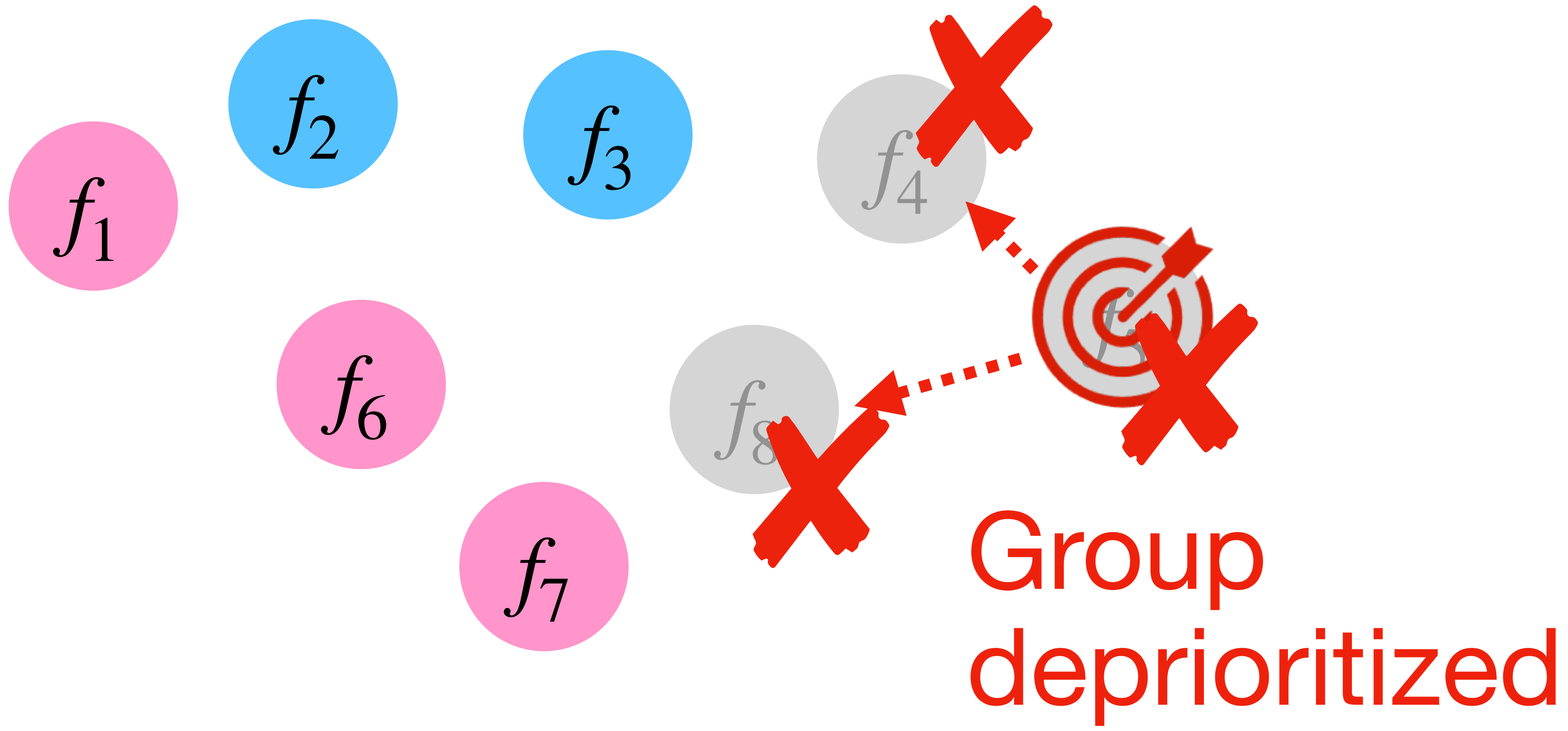
```
channelRead(x) {
  socket.read();
}


try {
  channelRead(…)
} catch(…) {
   log(“bad”)
}
```



Feedback Algorithm

How to update priority?

Priority 1: Trait priority



Idea: Less-explored trait should have higher priority

Trait o_k has priority O_k

Increments O_k for each unsuccessful trial

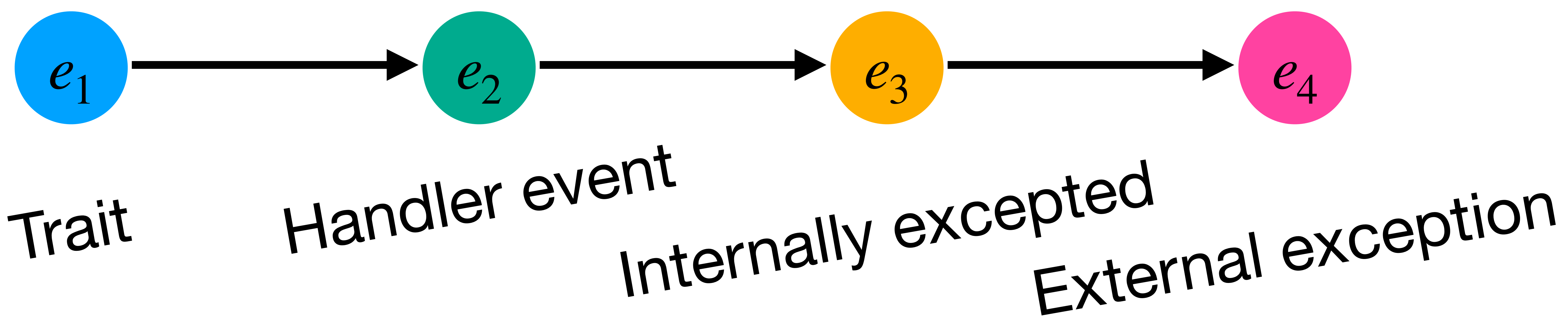
Smaller $O_k \Rightarrow$ higher priority

Feedback Algorithm

How to update priority?

Priority 2: Fault site priority

Ideal world:

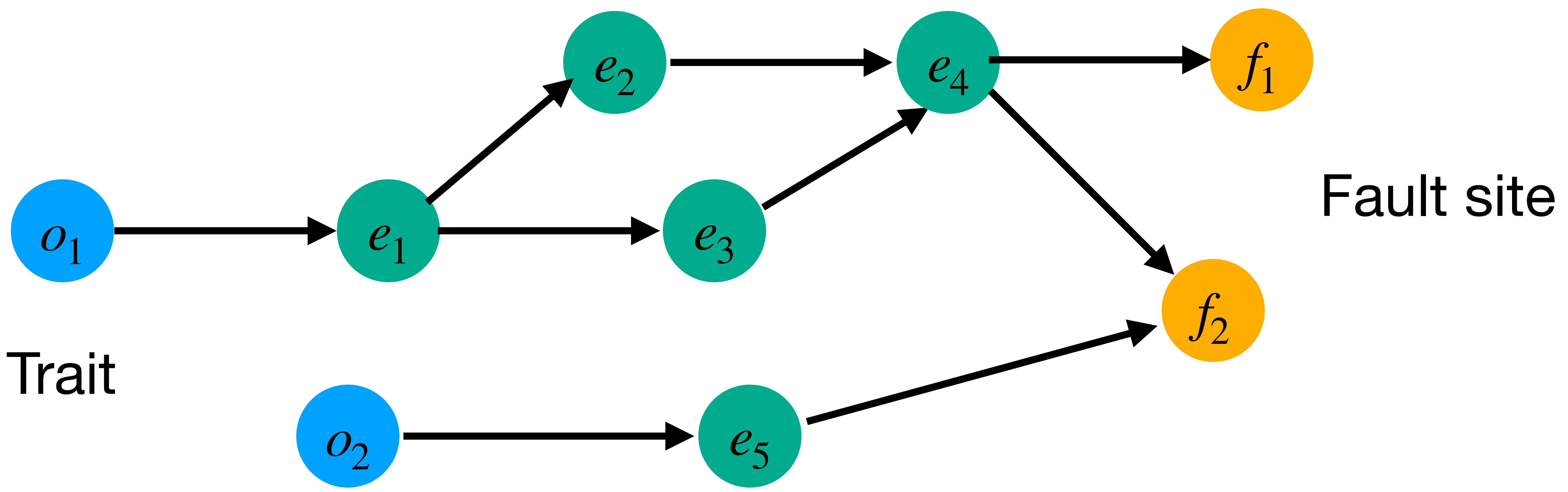


Feedback Algorithm

How to update priority?

Priority 2: Fault site priority

Reality:

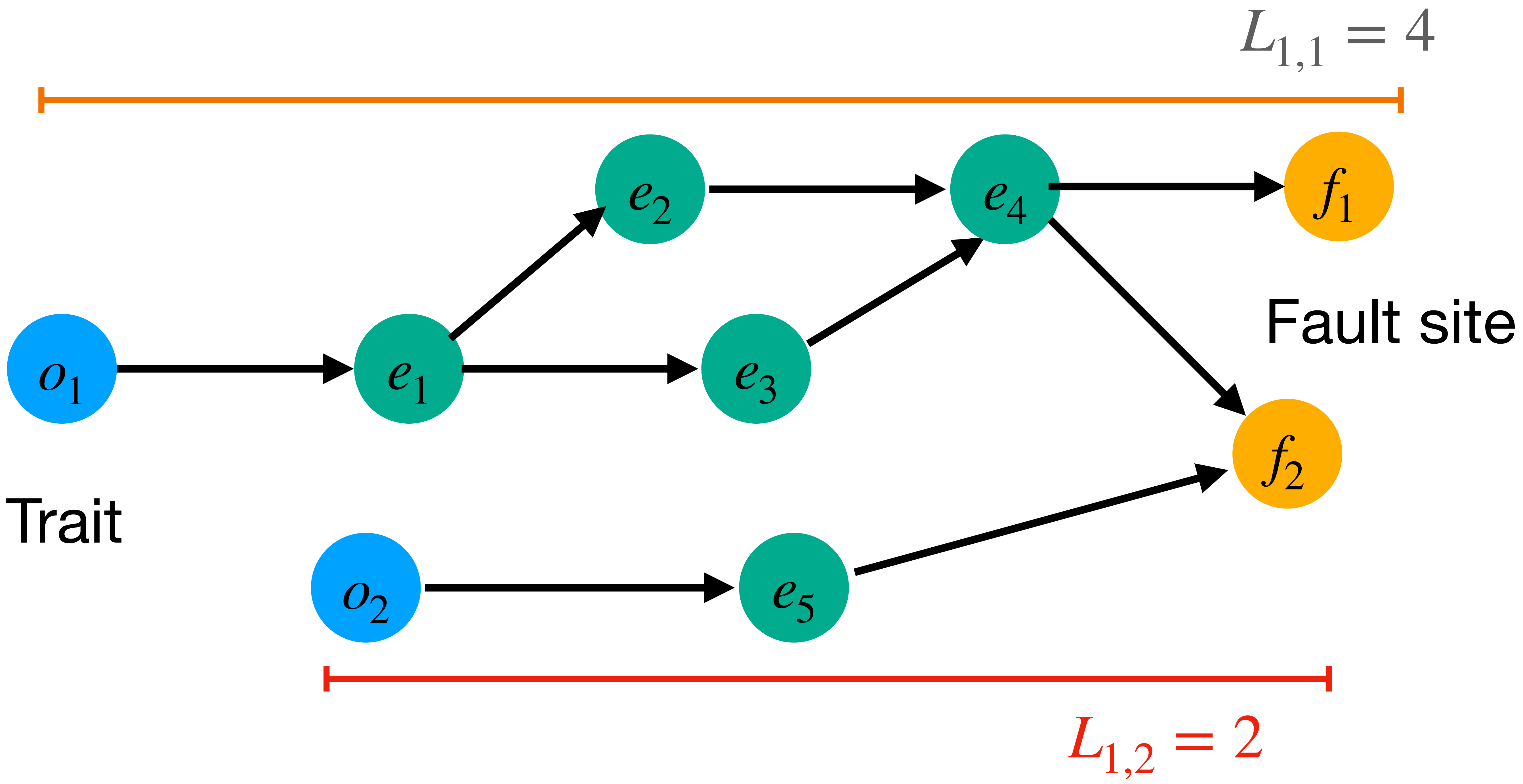


Problem: 1) trait & fault has uncertain mapping
2) branching wastes attempts

Feedback Algorithm

How to update priority?

Priority 2: Fault site priority



Introduce **spatial distance**

$L_{i,k}$

Shorter distance

⇒ More likely direct cause

⇒ Less wasted attempts

⇒ Faster exploration

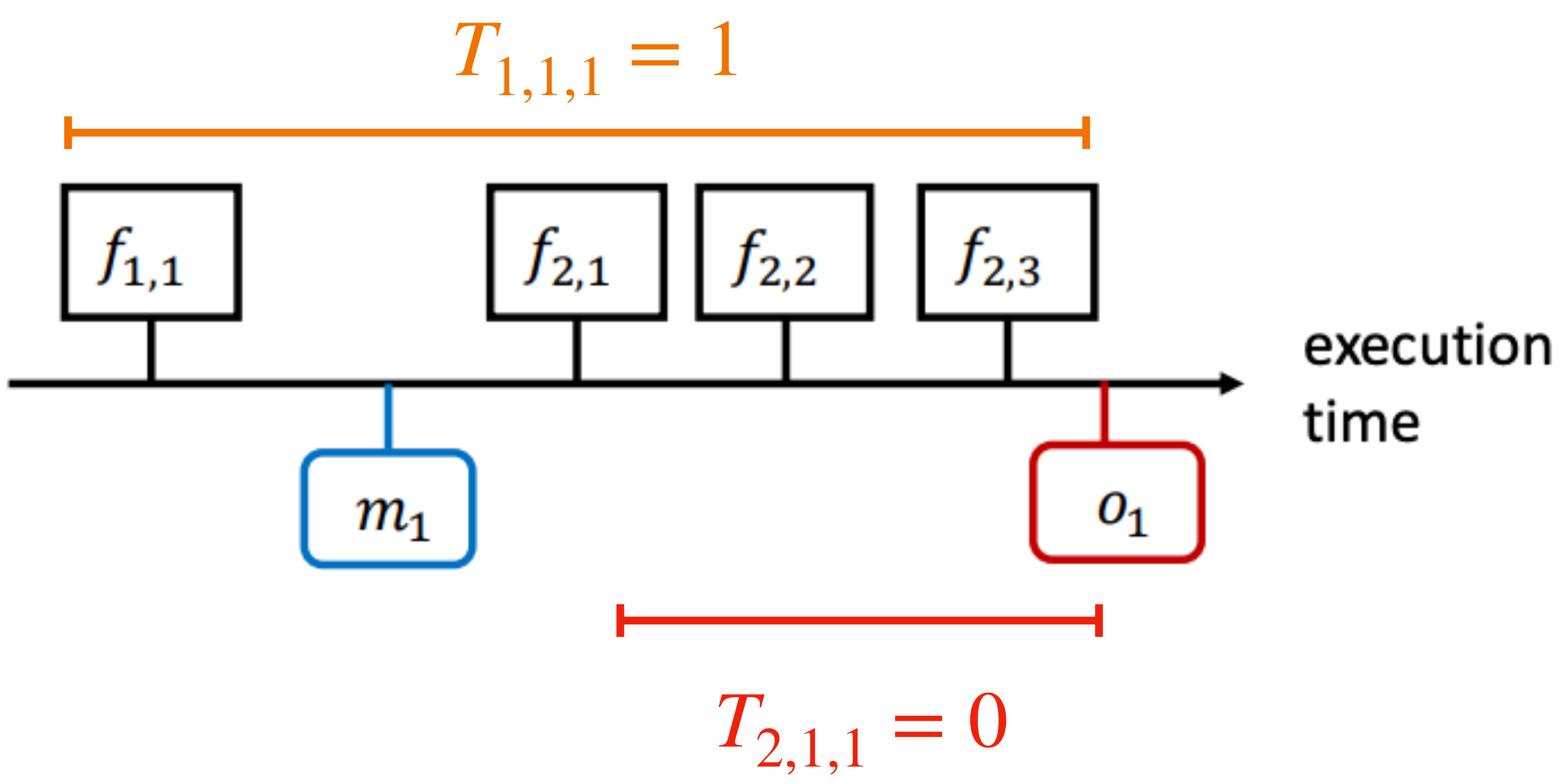
Feedback Algorithm

How to update priority?

Priority 3: Fault instance (timing) priority

Idea: The closer a fault in log, the more likely it is the cause

Introduce **logic time distance $\mathbf{T_{i,j,k}}$**: number of logs between the log and the fault



Check the paper for details!

Feedback Algorithm: Putting Together

User customizable priority formula using $O_k, L_{i,k}, T_{i,j,k}$

Anduril used priority:

$$\mathbf{F_{i,j}} = (min_k(O_k + L_{i,k}), T_{i,j,argmin_k(O_k+L_{i,k})})$$

Priority sorting: Primary: best fault site — Secondary: best occurrence

Feedback Algorithm: Putting Together

User customizable priority formula using $O_k, L_{i,k}, T_{i,j,k}$

Anduril used priority:

$$\mathbf{F_{i,j}} = (min_k(O_k + L_{i,k}), T_{i,j,argmin_k(O_k+L_{i,k})})$$

Priority sorting: Primary: best fault site Secondary: best occurrence

Updated by feedback loop

Experiment Setup

Evaluate on **5** large *real-world* distributed systems

Collect **40** *real-world* failures

Sample **22** for reproduction (I/O fault-related)

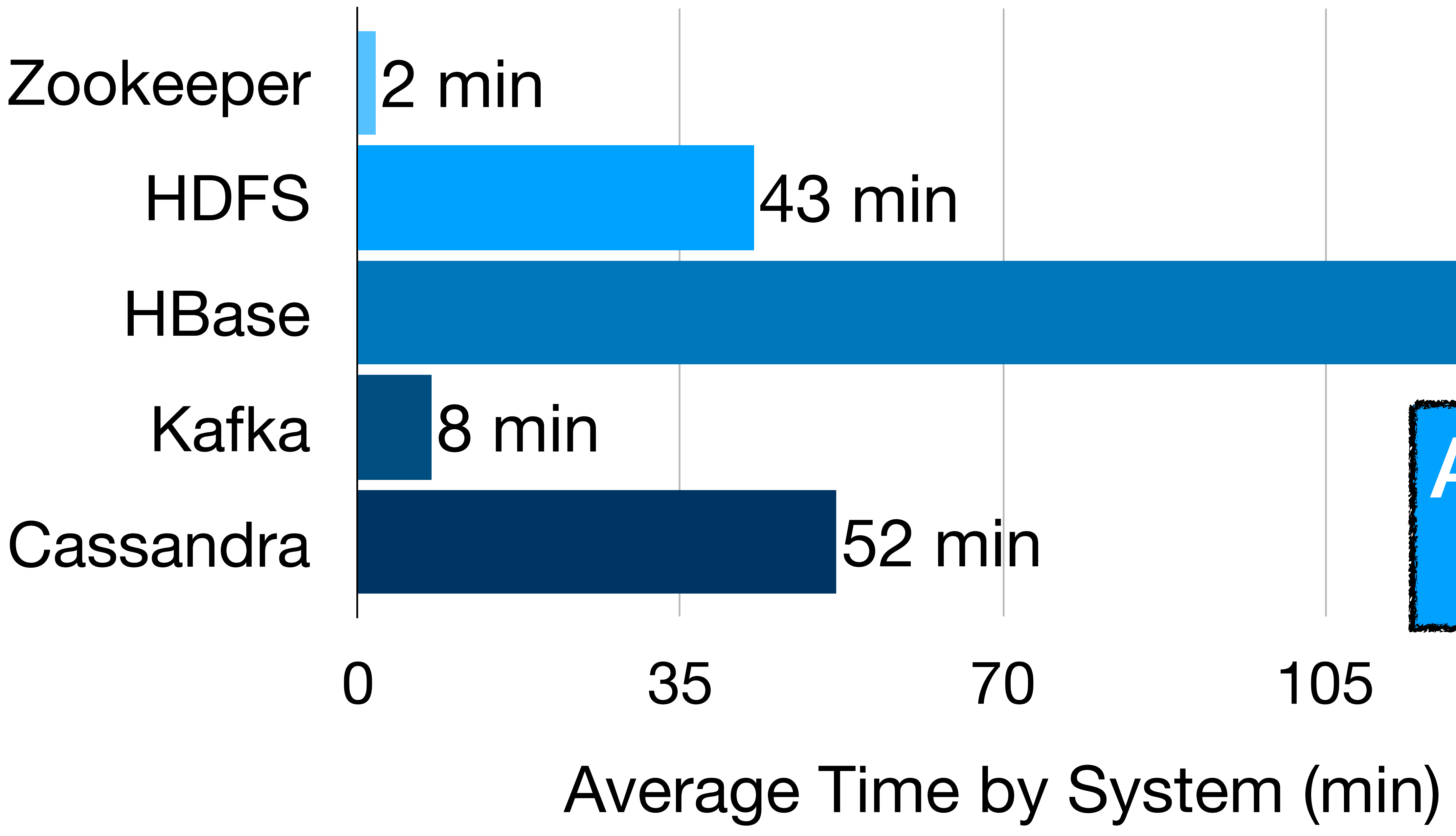
System	LOC	Fault sites	Fault instances	# Sampled
ZooKeeper	148 K	572	3 K	4
HDFS	769 K	4,761	73 K	7
HBase	930 K	2,905	106 K	6
Kafka	184 K	1,134	423 K	3
Cassandra	230 K	1,258	2023 K	2

Efficacy of Failure Reproduction

Effectiveness: Anduril reproduced **all 22** sampled cases

Efficiency: Anduril takes 2~445 minutes.

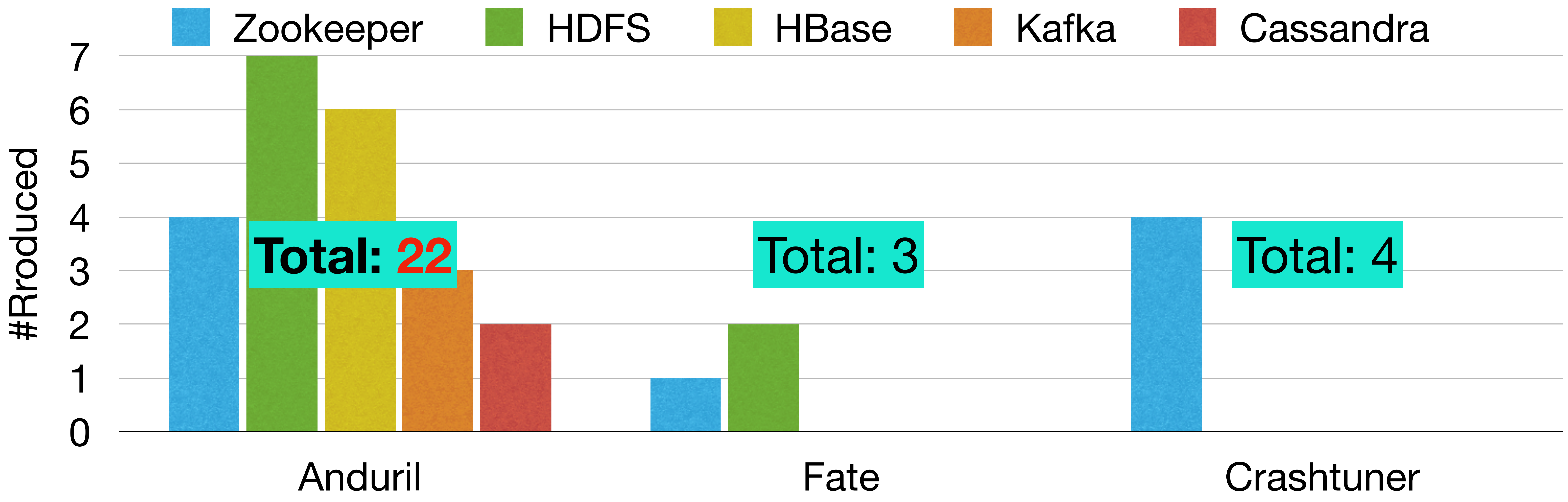
- For 6 cases with known developer effort: manually take ~136 hours.



Anduril is more efficient than manual

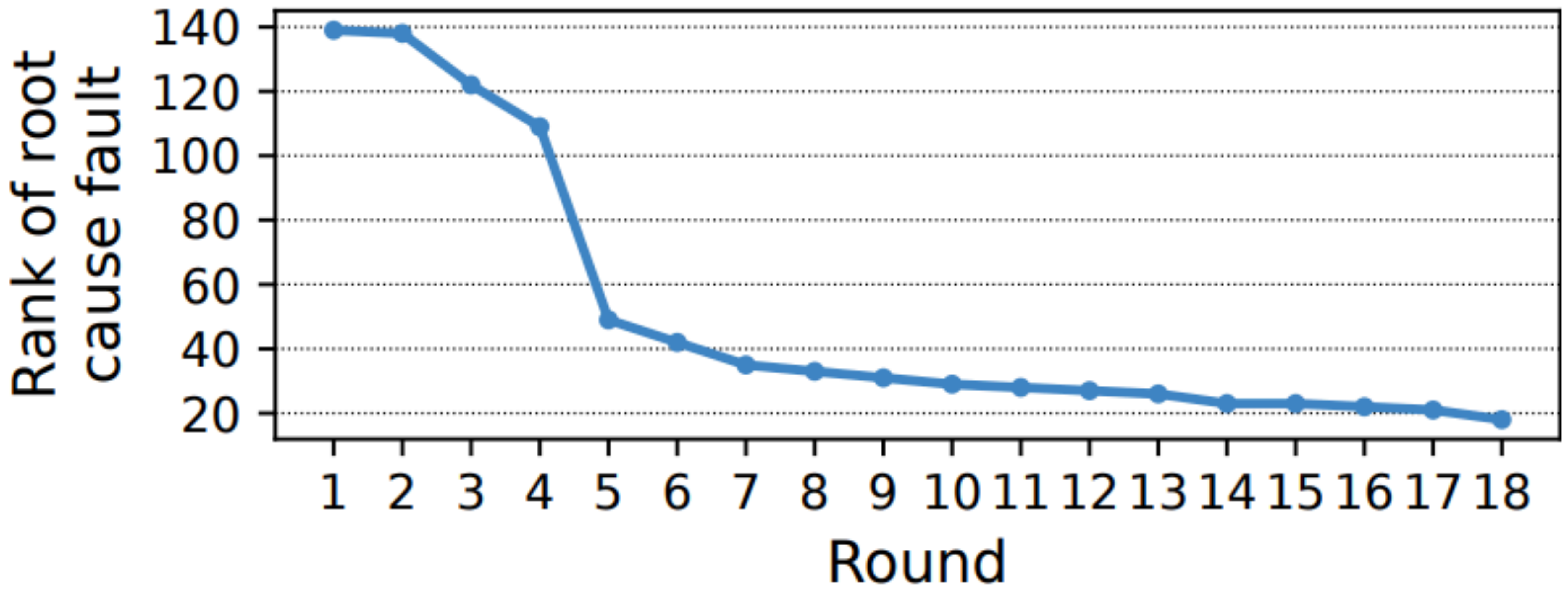
Evaluation: Comparison with SOTA

Compare with SOTA work **Fate** and **Crashtuner**



Anduril is more effective than SOTA

Evaluation: Effect of Feedback



It took 18 rounds to prioritize the root-cause fault's rank from initial 140 to 20

Conclusion

Anduril: A fault injection tool designed to efficiently reproduce fault-induced failures in deployed distributed systems

- Use static causal reasoning to prune fault sites
- Use a novel dynamic feedback-driven injection algorithm to search for the root-cause fault and timing in a large fault space

https://github.com/OrderLab/Anduril