Understanding and Detecting Software Upgrade Failures in Distributed Systems

By Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Shan Lu, Ding Yuan

Presented by Andrey Satarin, @asatarin
September, 2022

https://asatarin.github.io/talks/2022-09-upgrade-failures-in-distributed-systems/

Outline

- Introduction
- Findings on Severity and Root Causes
- Testing and Detecting
- Conclusions
- Personal Experience and Commentary

Introduction

Software upgrade failures

Software upgrade failures — failures that **only occur during software upgrade**. Never occur under regular execution scenarios.

- **Not** failure-inducing configurations change
- **Not** bug in only new version of software

Defects from **two versions** of software **interacting**

Why upgrade failures are important?

- **Large scale** — touches the whole system or large part
- Vulnerable context — upgrade is a **disruption in itself**
- Persistent Impact — can **corrupt persistent data irreversibly**
- **Difficult to expose in house** — little focus in testing

What was studied?

- Symptoms and severity
- Root causes
- Triggering conditions
- Ways to detect upgrade failures

Number of upgrade failures analyzed

- Cassandra — 44
- HBase — 13
- HDFS — 38
- Kafka — 7
- MapReduce — 1
- Mesos — 8
- Yarn — 8
- ZooKeeper — 4

Total: 123 bugs

Findings on Severity and Root Causes

Finding 1

Upgrade failures have **significantly higher priority** than regular failures

Larger share of bugs is high priority compared to non-upgrade failures

Finding 2

The **majority (67%) of upgrade failures are catastrophic** (i.e., affecting all or a majority of users instead of only a few of them). This percentage is much higher than that (24%) among all bugs

- 28% bring down the **entire cluster**
- Catastrophic **data loss** or performance degradation

Finding 3

Most (70%) upgrade failures have **easy-to-observe symptoms** like node crashes or fatal exceptions

Finding 4

The majority (63%) of upgrade bugs were **not caught before code release**

=> We need to **get better at testing** upgrades

Finding 5

About **two thirds** of upgrade failures are caused by interaction between two software versions that hold **incompatible data syntax or semantics** assumption

Out of those:

- 60% in persistent data and 40% in network messages
- 2/3 syntax difference and 1/3 semantic difference

Finding 6

Close to 20% of syntax incompatibilities are **about data syntax** defined by serialization libraries or enum data types. Given their clear syntax definition interface, **automated incompatibility detection is feasible**

Finding 10

All of the upgrade failures require **no more than 3 nodes** to trigger

[OSDI14] "Simple Testing Can Prevent Most Critical Failures":

"Finding 3. **Almost all (98%)** of the failures are guaranteed to manifest on **no more than 3 nodes**. 84% will manifest on no more than 2 nodes."

Finding 11

Close to 90% of the upgrade failures are **deterministic**, not requiring any special timing to trigger

Testing and Detecting

Limitations in state of the art

(As presented in the paper)

- Do not solve problem of **workload generation**
  - Testing workloads are **designed from scratch (BAD!)**
- No mechanism **to systematically explore** different version combinations, configuration or update scenarios

DUPTester

DUPTester

- DUPTester — **D**istributed system **UP**grade **T**ester
- Simulates **3-node cluster** with containers
- Systematically tests **three scenarios**:
  - Full-stop upgrade
  - Rolling upgrade
  - Adding new node

Testing workloads

From section 6.1.2 Testing workload:

"As discussed in Section 5.3, a **main challenge** facing all existing systems is to **come up with workload** for upgrade testing"

DUPTester:

- Using stress testing is straightforward
- Using "unit" testing requires some tricks

Using “unit” tests as workload

Two strategies:

- Automatically **translate** “unit” tests into client-side scripts
  - Not guaranteed to translate everything
  - Needs function mapping from developers
- Execute on V1 and **successfully start on V2**

DUPChecker

DUPChecker

Types of syntax incompatibilities:

- Serialization libraries definition syntax incompatible across versions
    - Open source alternatives exist
- Incompatibility of Enum-typed data

DUPChecker

Serialization libraries:

- Parses protobuf definitions
- Compares them across versions to find incompatibilities

Enums:

- Data flow analysis to find persisted enums
- Check if enum index is persisted and there are additions/deletions in enum

Conclusions

Conclusions

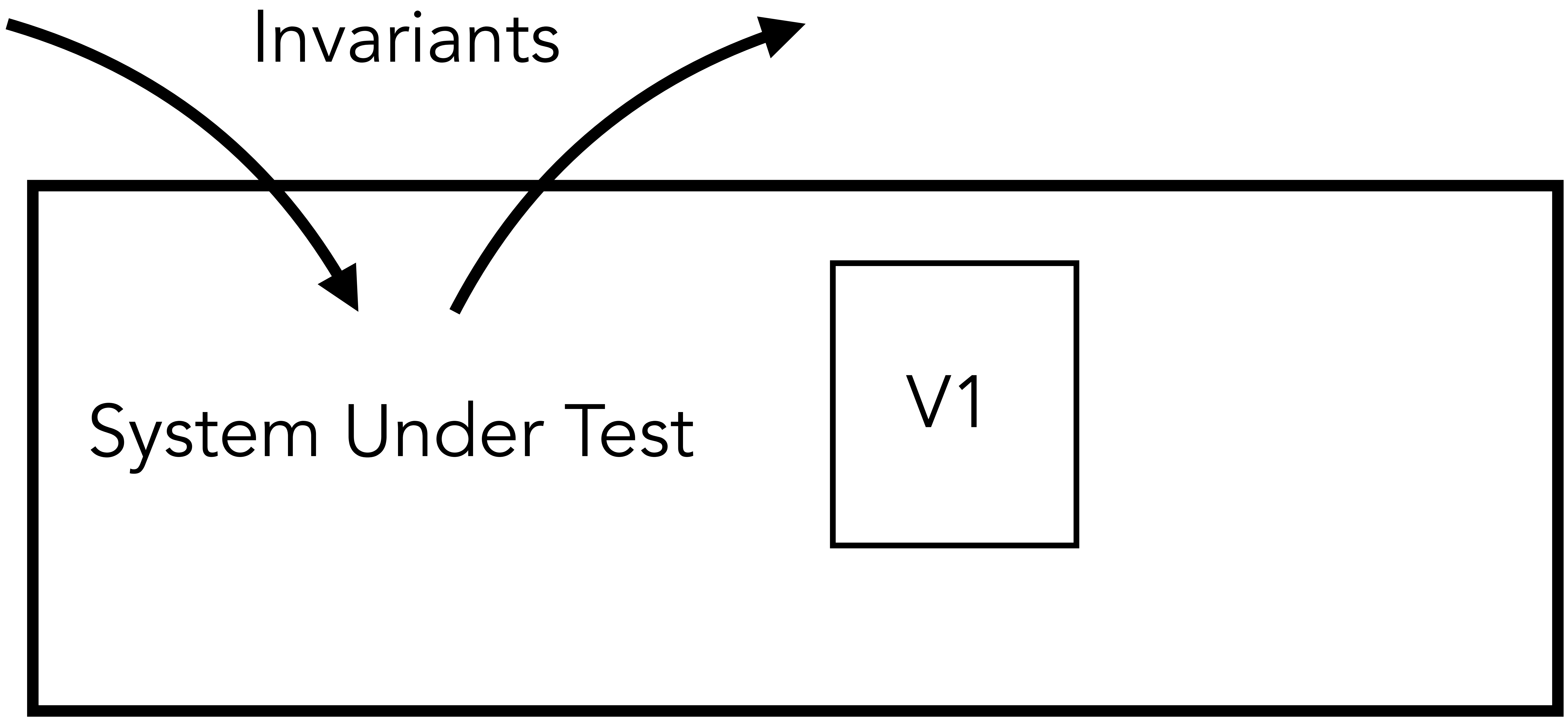
- **First in-depth analysis** of upgrade failures
- Upgrade failures have **severe consequences**
- DUPTester **found 20 new upgrade failures** in 4 systems
- DUPChecker detected **800+ incompatibilities** in 7 systems
- Apache HBase team requested DUPChecker **to be a part of their pipeline**

Personal Experience and Commentary

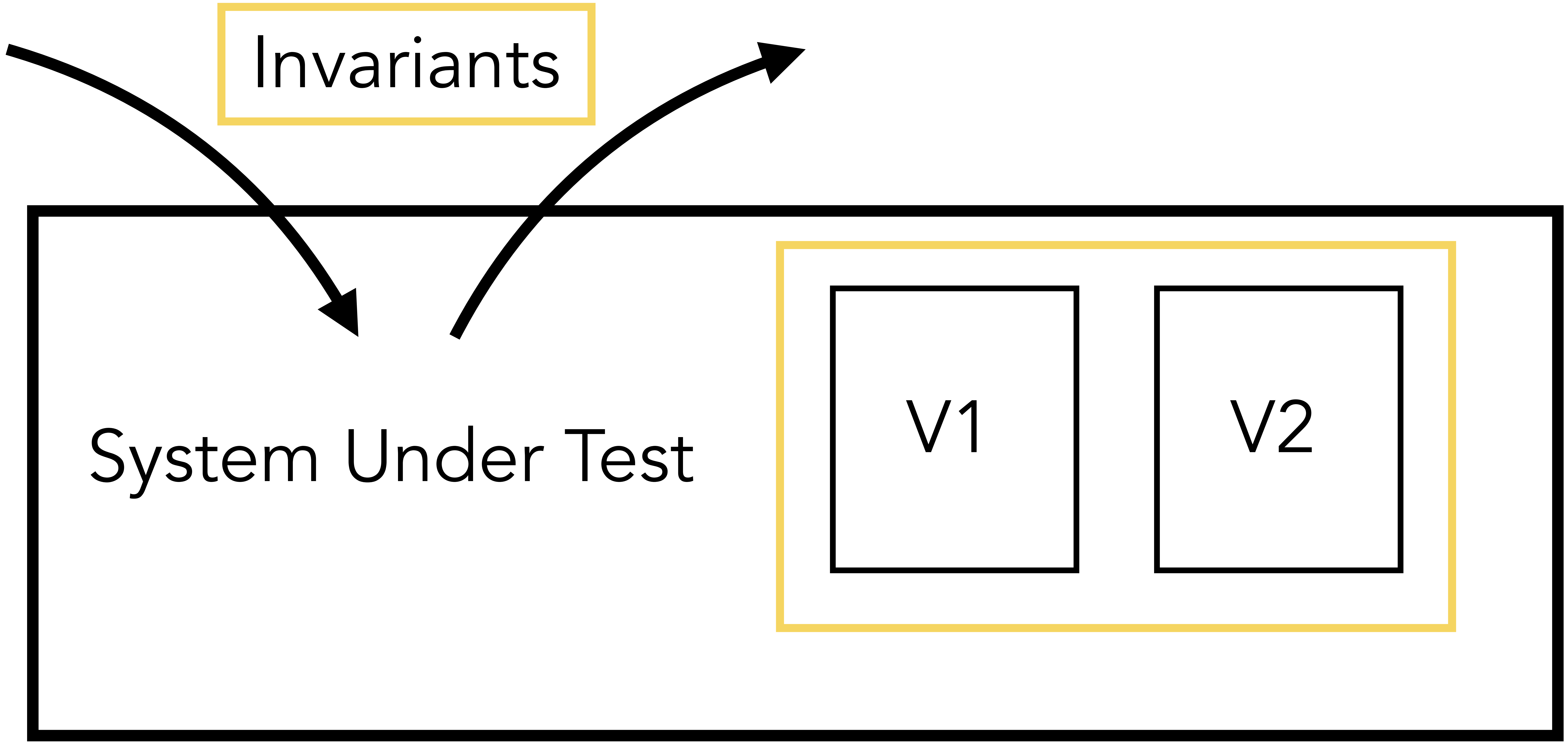
Upgrades and correctness

- Stress tests usually **do not include correctness validation**
- Correctness implies correctness **with failure injection**
- Testing system upgrade implies testing **rollback**

System as a black box



System as a black box



Testing workload

From section 6.1.2 Testing workload:

> "As discussed in Section 5.3, a **main challenge** facing all existing systems is to **come up with workload** for upgrade testing"

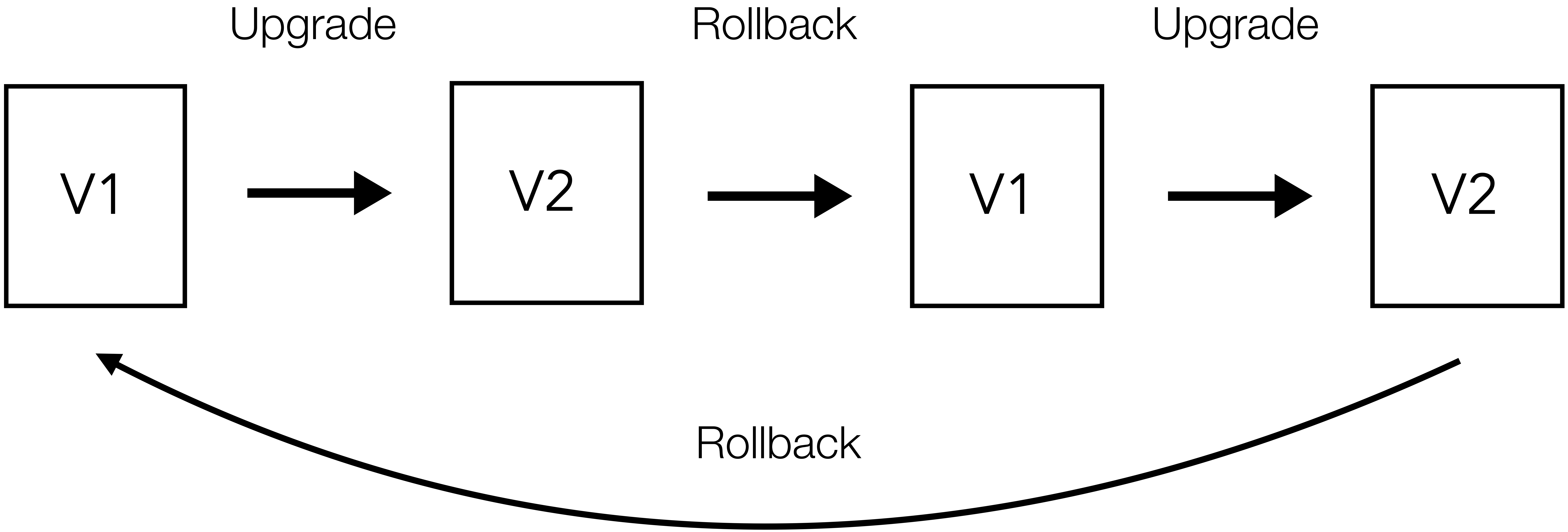
You probably already have workloads to test correctness:

- Stress tests
- **Correctness tests** (probably Jepsen-like) [Jepsen22]

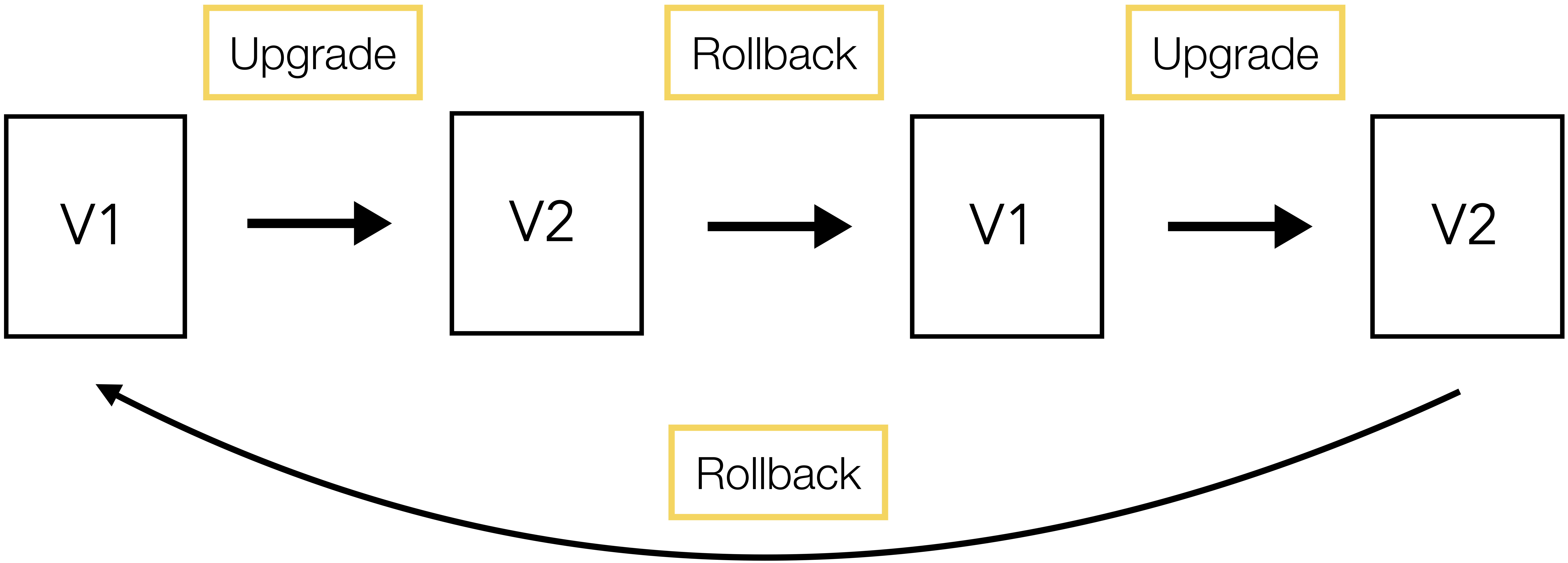
Upgrade and rollback

- We need to test both **upgrade and rollback**
- Both operations ideally tested with **failure injection**
- **Probability of exposing bugs** ~ “mixed version time”
- We should **maximize “mixed version time”**

Upgrade and rollback



Upgrade and rollback



Conclusions (2)

- There is certainly value in research and ideas from the paper
- There are additional ways one can **improve upgrade testing** by leveraging correctness tests
- System during upgrade == system during normal operation

Thank you for your attention

References

- Self reference for this talk (slides, video, etc) https://asatarin.github.io/talks/2022-09-upgrade-failures-in-distributed-systems
- "Understanding and Detecting Software Upgrade Failures in Distributed Systems" paper https://dl.acm.org/doi/10.1145/3477132.3483577
- Talk at SOSP 2021 https://youtu.be/29-isLcDtL0
- Reference repository for the paper https://github.com/zlab-purdue/ds-upgrade

References

- [OSDI14] Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

- [Jepsen22] https://jepsen.io/

Contacts

- Follow me on Twitter @asatarin
- Other public talks https://asatarin.github.io/talks/
- https://www.linkedin.com/in/asatarin/