Why Do Upgrades Fail and What Can We Do About It?

Tudor Dumitraș
Priya Narasimhan

Carnegie Mellon University

Upgrades in Enterprise Systems

Problem: – software upgrades are unreliable
– increasing cost of unplanned downtime

- AT&T Wireless (2003): outages, data-loss, $100M loss
- Hospital system (2006): medication unavailable in ER

Goal: dependable, online end-to-end upgrades

Approach: eliminate leading cause of upgrade failure

Current Approaches

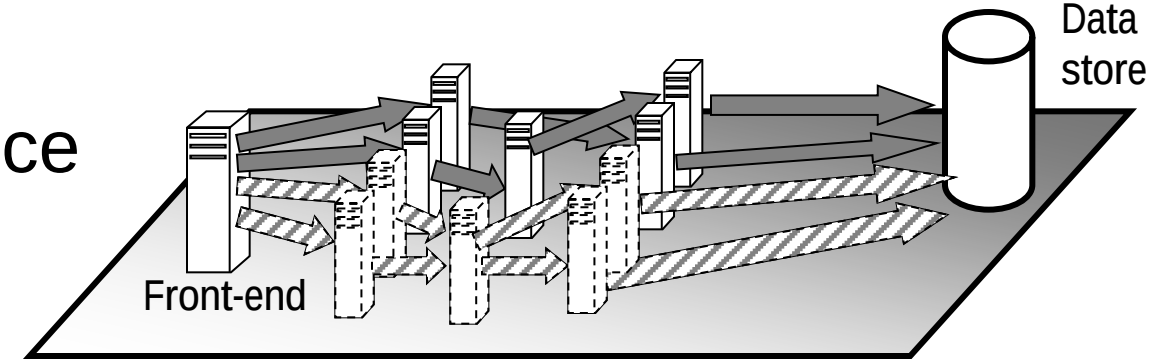
- Previous research
  - Dynamic software updating: [Segal & Frieder, 1993] [Neamtiu+, 2006]
  - Database-schema evolution: [Ferrandina+, 1995] [Curino+, 2008]
  - Distributed-system upgrades: [Bloom, 1983] [Kramer & Magee, 1985] [Ajmani+, 2006]

update running programs

migrate data offline

upgrade distributed objects/components

- Upgrades in practice
  - Rolling upgrades



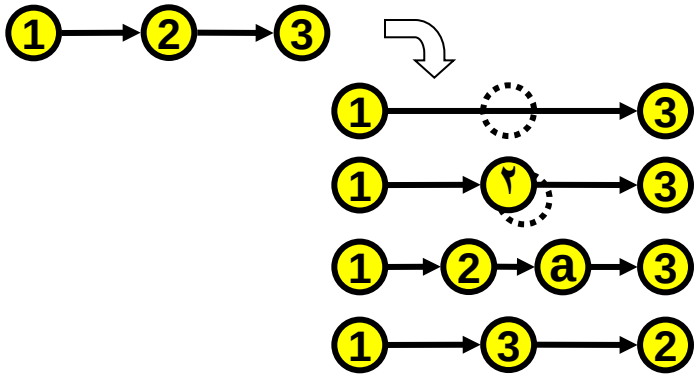
Outline

- **Why do upgrades fail …**
  - Upgrade-centric fault model
- **… and what can we do about it?**
  - Imago: dependable, online upgrades
  - Dependability evaluation

Upgrade Faults

Most upgrade failures: procedure violations [Crameri+, SOSP'07]

- Procedure violations
  - Omitted action
  - Incorrect action
  - Spurious action
  - Order inversion



- Three sources of upgrade-fault data
  - User study [Nagaraja+, 2004]
  - Survey [Oliveira+, 2006]
  - Field study (Apache bug reports from 2007)

Procedure violations occur in 43% of cases

Hidden Dependencies

Cannot be detected automatically
or
are overlooked owing to their complexity

Hidden-Dependency Examples

Service location
- File path
- Network address

Dynamic linking
- Library conflicts
- Defective components

Database schema
- Application/DB mismatch
- Missing indexes

Access privileges
(excessive / insufficient / unavailable)
- File system
- Database objects
- URLs

Configuration-parameter constraints

Replication degree

Storage-space availability

Client access to system-under-upgrade

Cached data
- SSL certificates
- DNS lookups
- Buffer cache

Listening-port conflicts

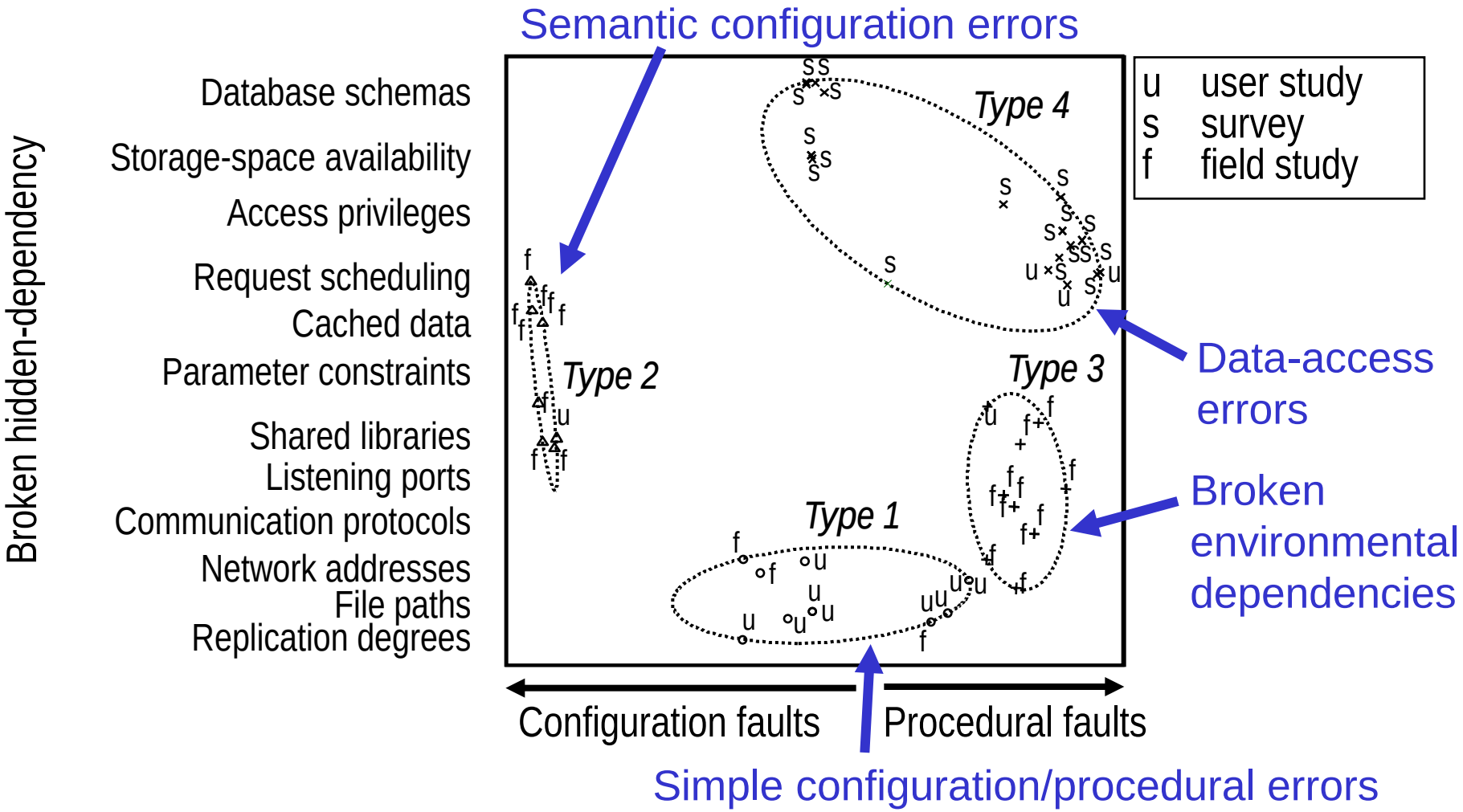
Protocol mismatch

Entropy for random-number generation

Request scheduling

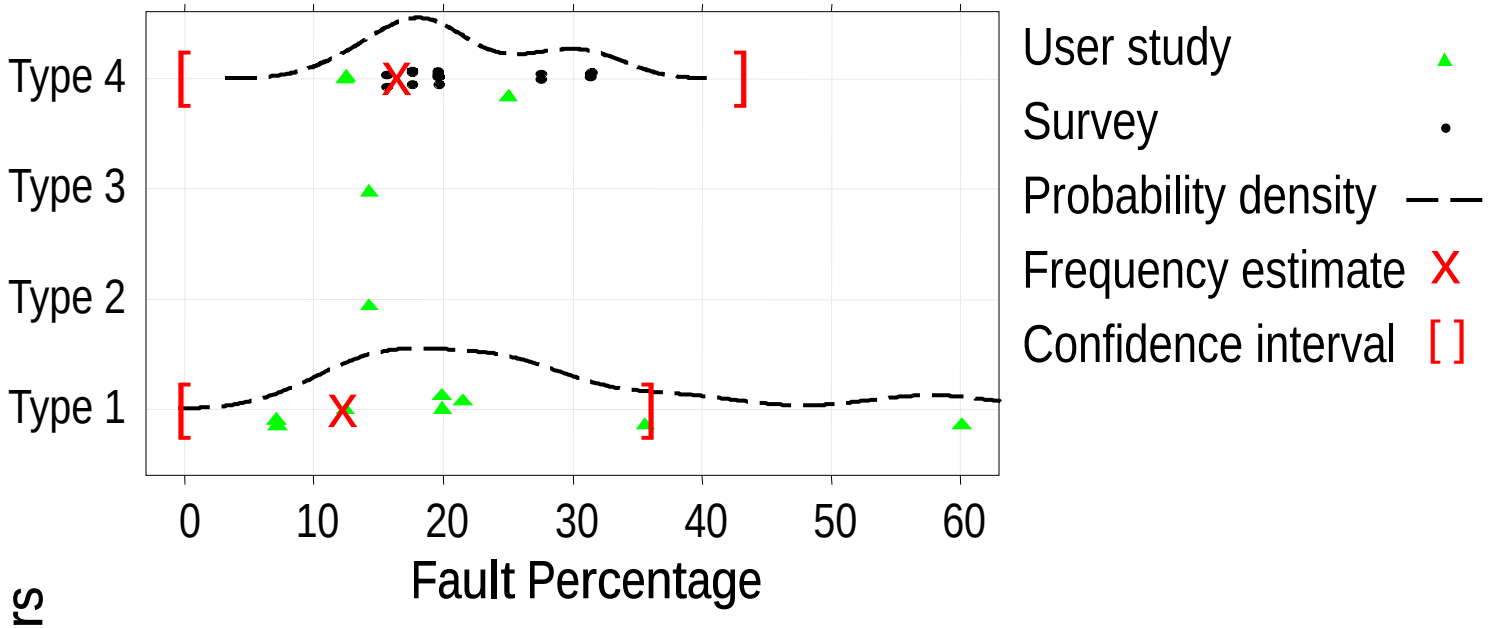
Disk speed

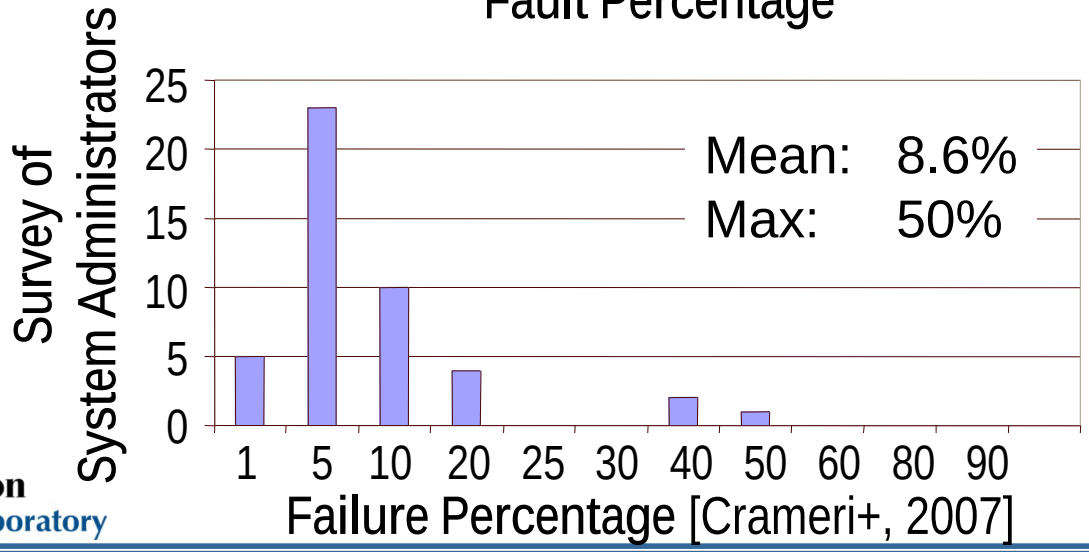
Upgrade-Centric Fault Model



http://www.ece.cmu.edu/~tdumitra/upgrade_faults

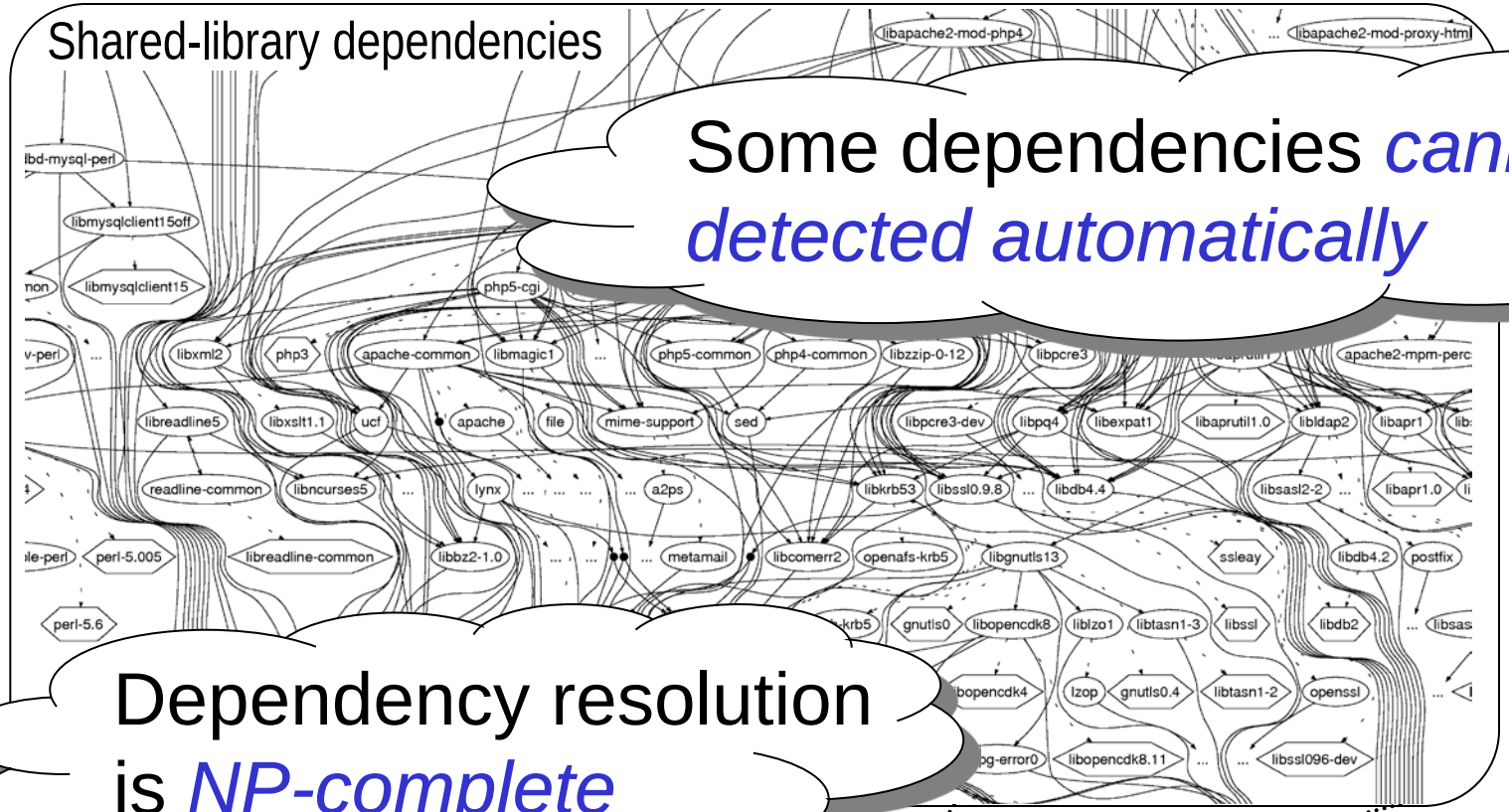
Upgrade-Fault Frequencies





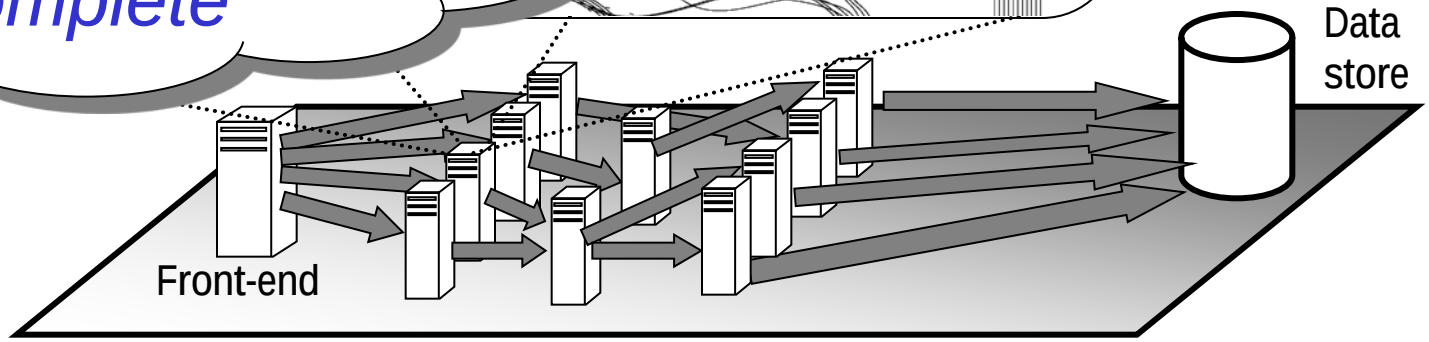
Carnegie Mellon
Parallel Data Laboratory

Why Do We Break Dependencies?



Some dependencies *cannot be detected automatically*

Dependency resolution is *NP-complete*



Carnegie Mellon
Parallel Data Laboratory

Outline

- **Why do upgrades fail …**
  - Upgrade-centric fault model

- **… and what can we do about it?**
  - Imago: dependable, online upgrades
  - Dependability evaluation

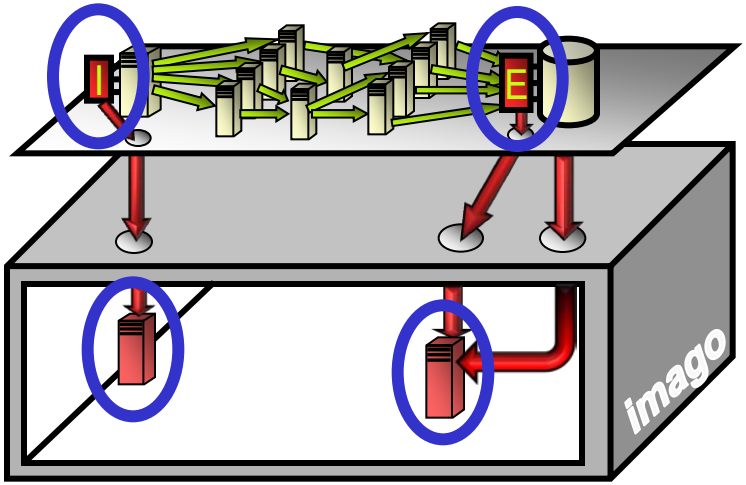
What Do We Want?

- Isolation: the upgrade must not affect the dependencies of the online version
- Atomicity: the full functionality of either the old or the new version must be available
- Fidelity: the testing and production environments must be identical

Imago

Assumptions

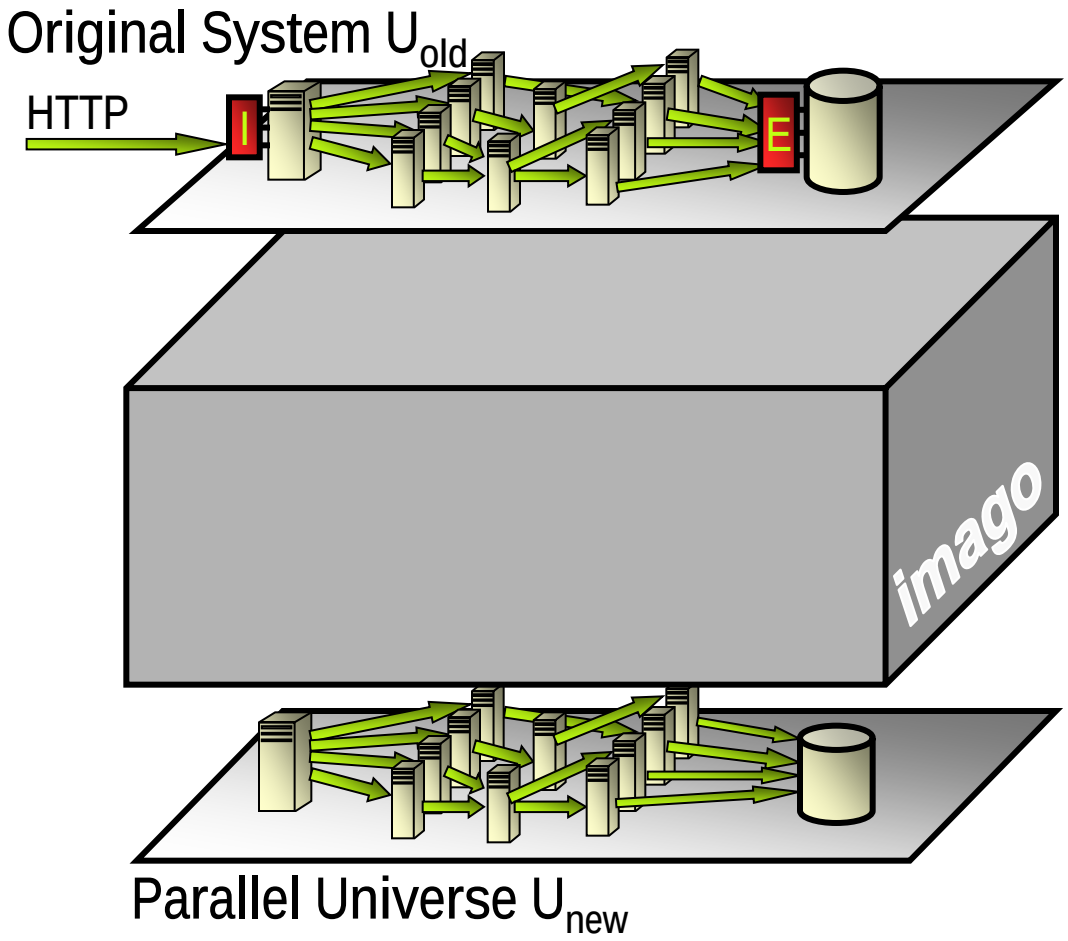
- Well-defined, static ingress and egress points
- Read-mostly workload
- System-under-upgrade provides hooks:
  - Flushing in-progress updates
  - Reading data without obstructing live requests



Components

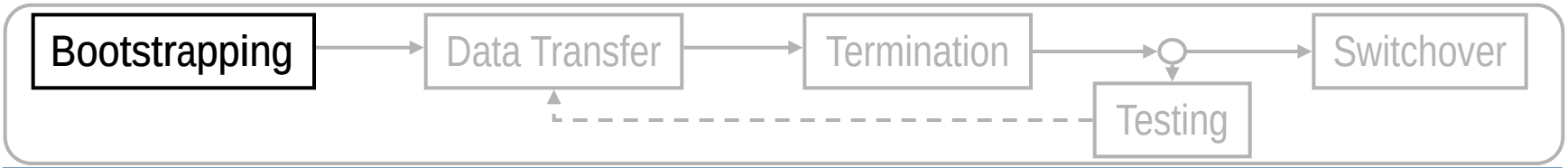
- Interceptors
  - EGRESS: To storage
  - INGRESS: From clients
- Upgrade driver
- Compare engine

Imago: Bootstrapping

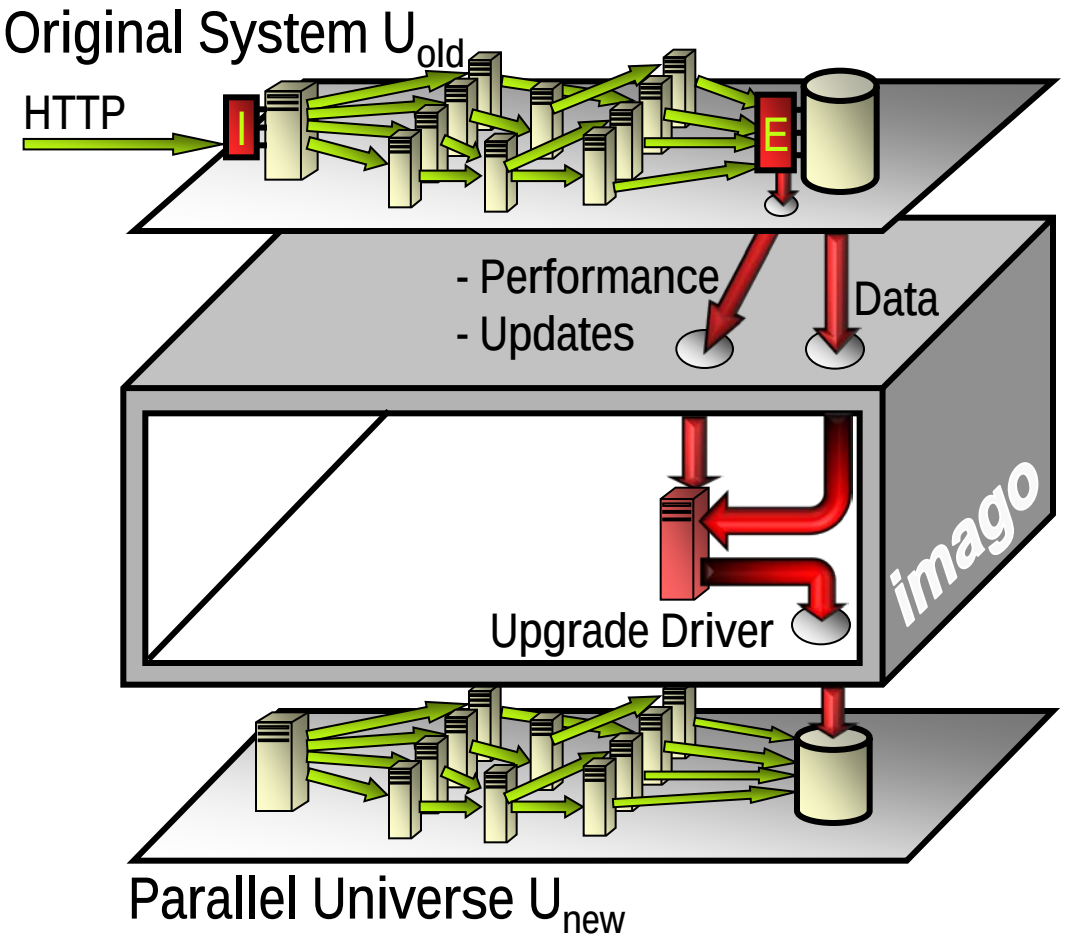


Install new version in U_{new}

- Separate hardware, or
- Virtual infrastructure



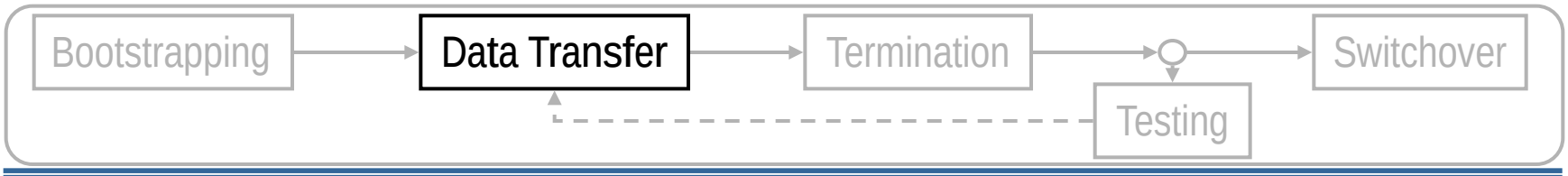
Imago: Data Transfer



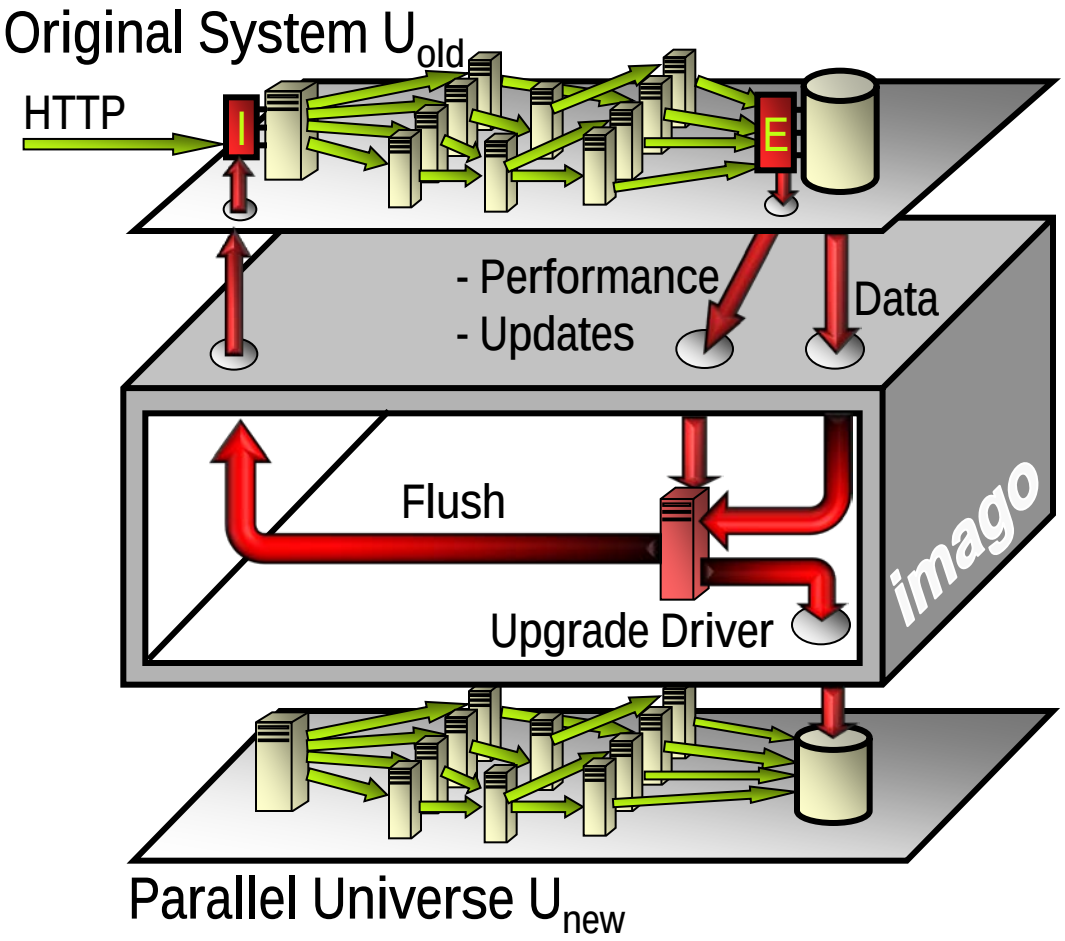
Install new version in U_{new}

Transfer data

- Opportunistic protocol
- Monitor updates at E
- Adapt to live workload



Imago: Termination

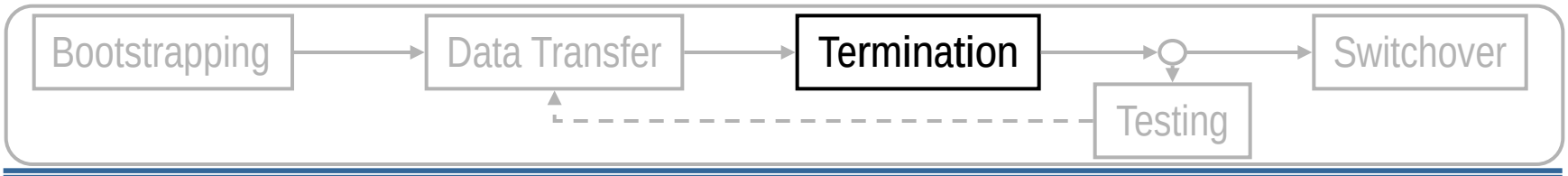


Install new version in U_{new}

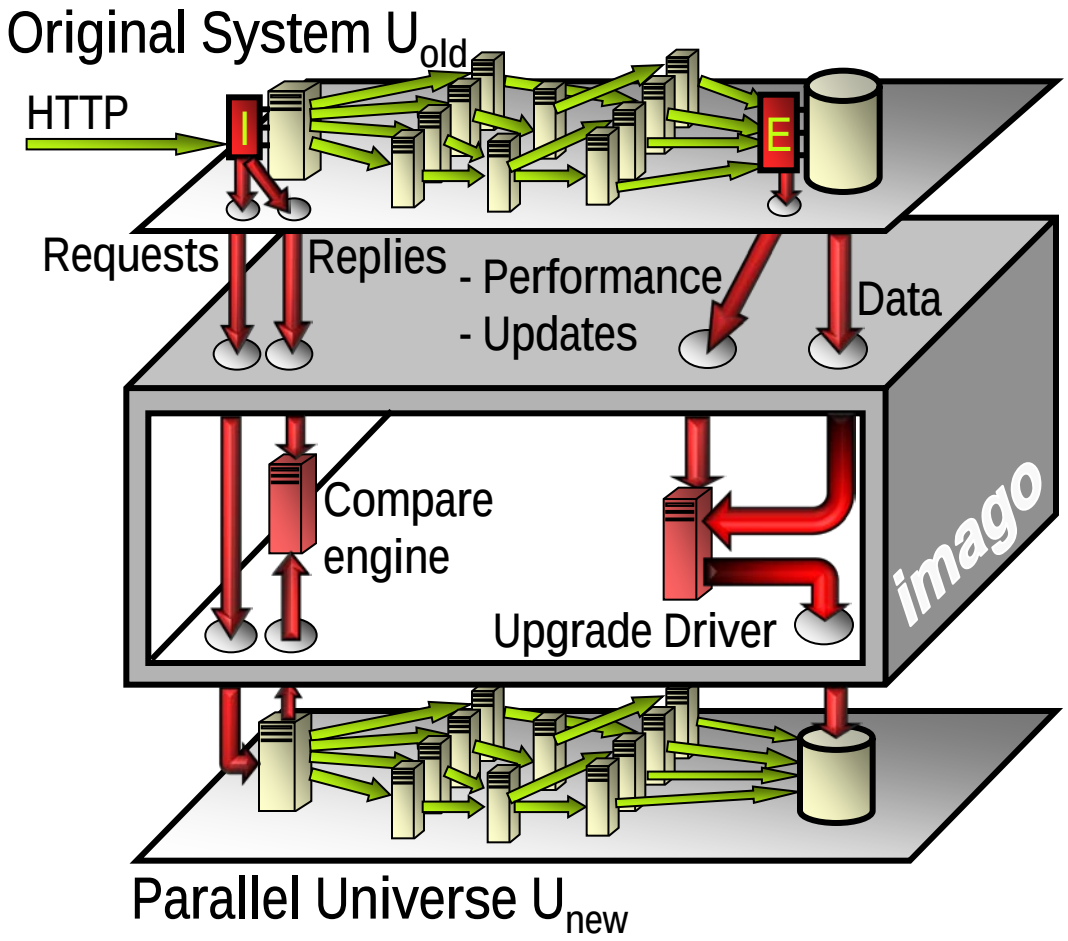
Transfer data

Flush in-progress requests

- Impose quiescence at I
- Flush updates atomically



Imago: Testing



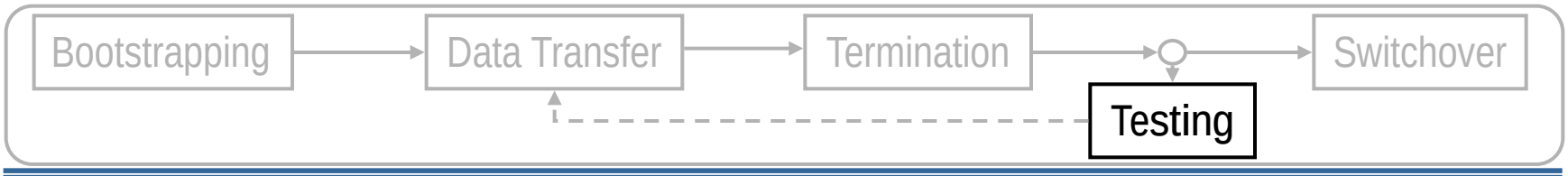
Install new version in U_{new}

Transfer data

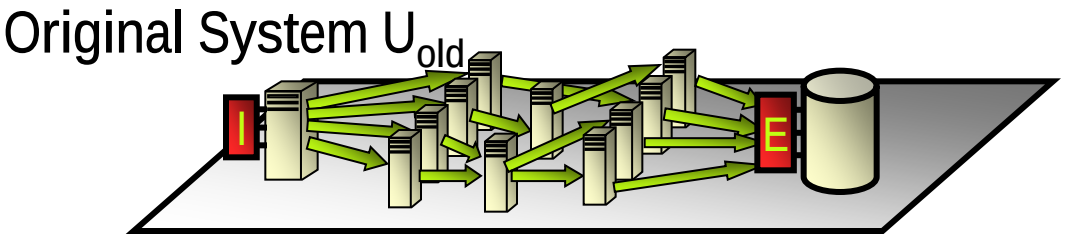
Flush in-progress requests

Test upgrade

- Unit and integration tests
- Operational tests
  - Using live workload



Imago: Switchover



Install new version in U_{new}

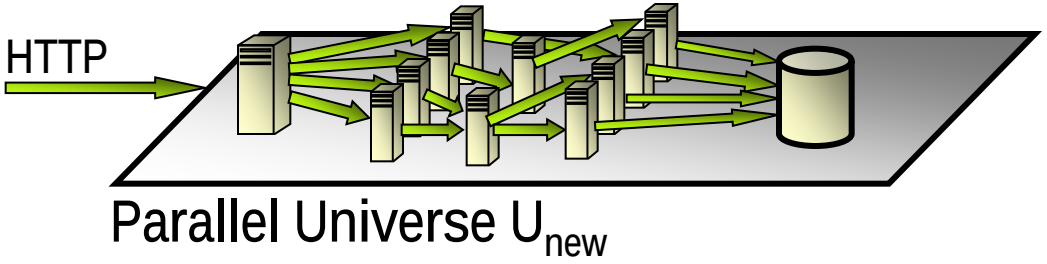
Transfer data

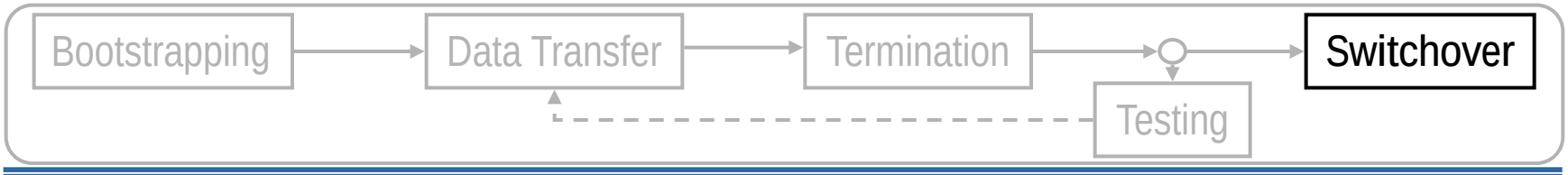
Flush in-progress requests

Test upgrade

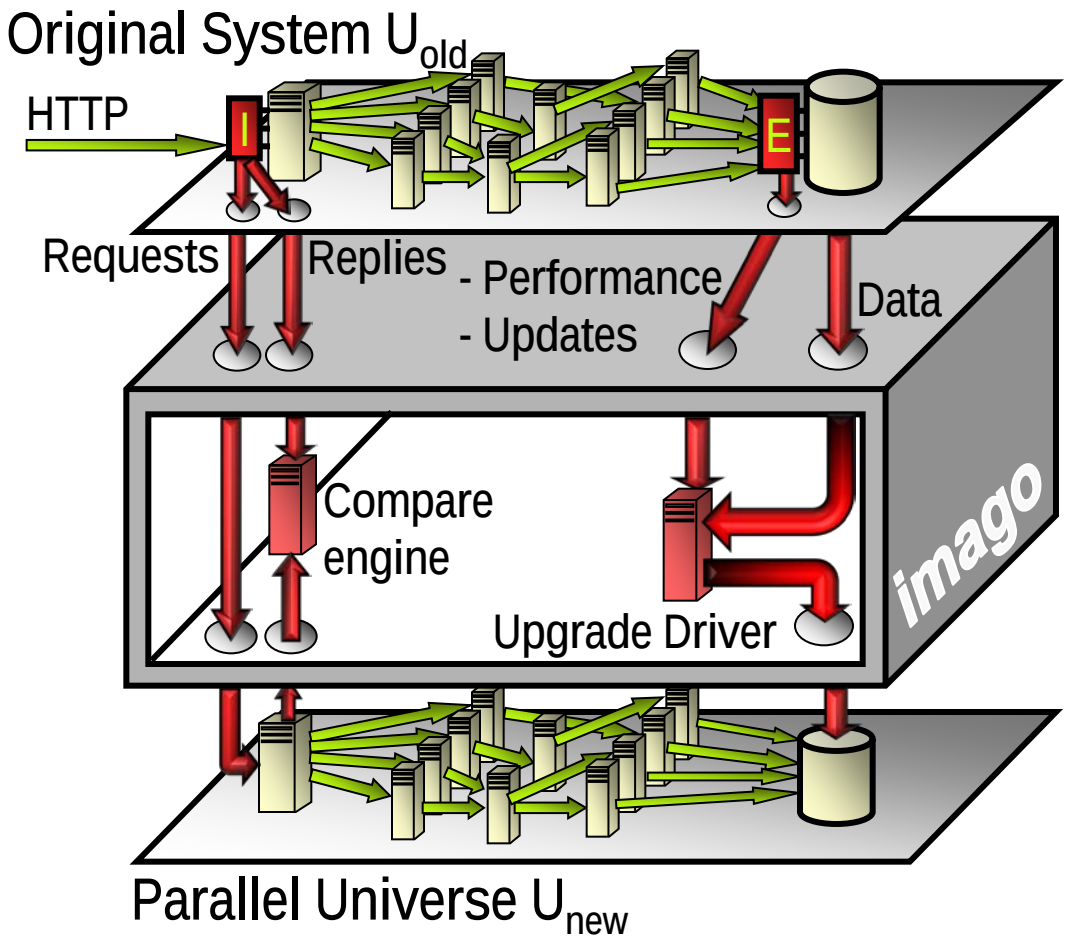
Switch-over atomically

- Redirect all traffic to U_{new}





Imago: Summary



Isolation

Install new version in U_{new}

Transfer data

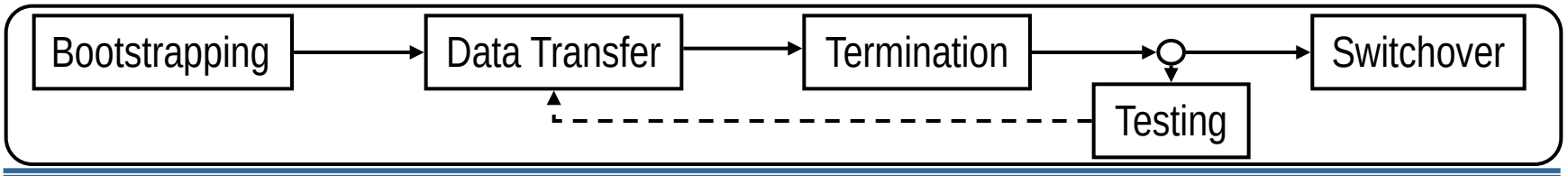
Flush in-progress requests

Test upgrade

Fidelity

Switch-over atomically

Atomicity

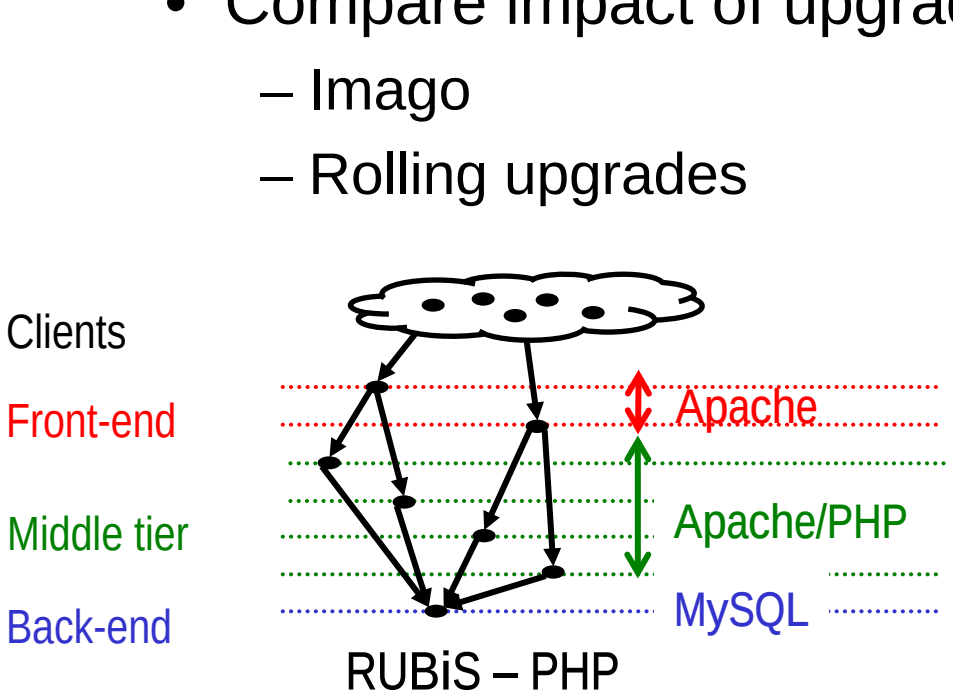


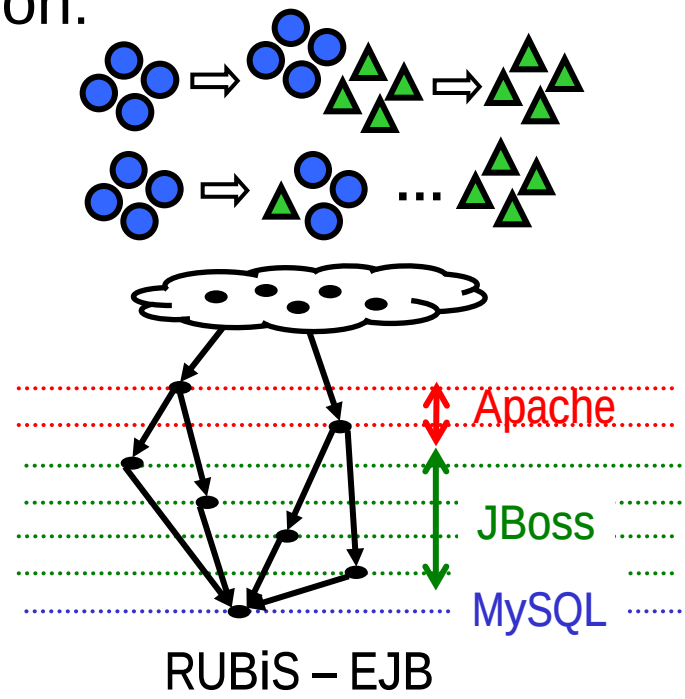
Outline

- **Why do upgrades fail …**
  - Upgrade-centric fault model
- **… and what can we do about it?**
  - Imago: dependable, online upgrades
  - Dependability evaluation

Experimental Methods

- What overhead does Imago impose?
  - Runtime overhead, switchover duration
- Does Imago reduce the upgrade failures?
  - Compare impact of upgrade faults on:
    - Imago
    - Rolling upgrades

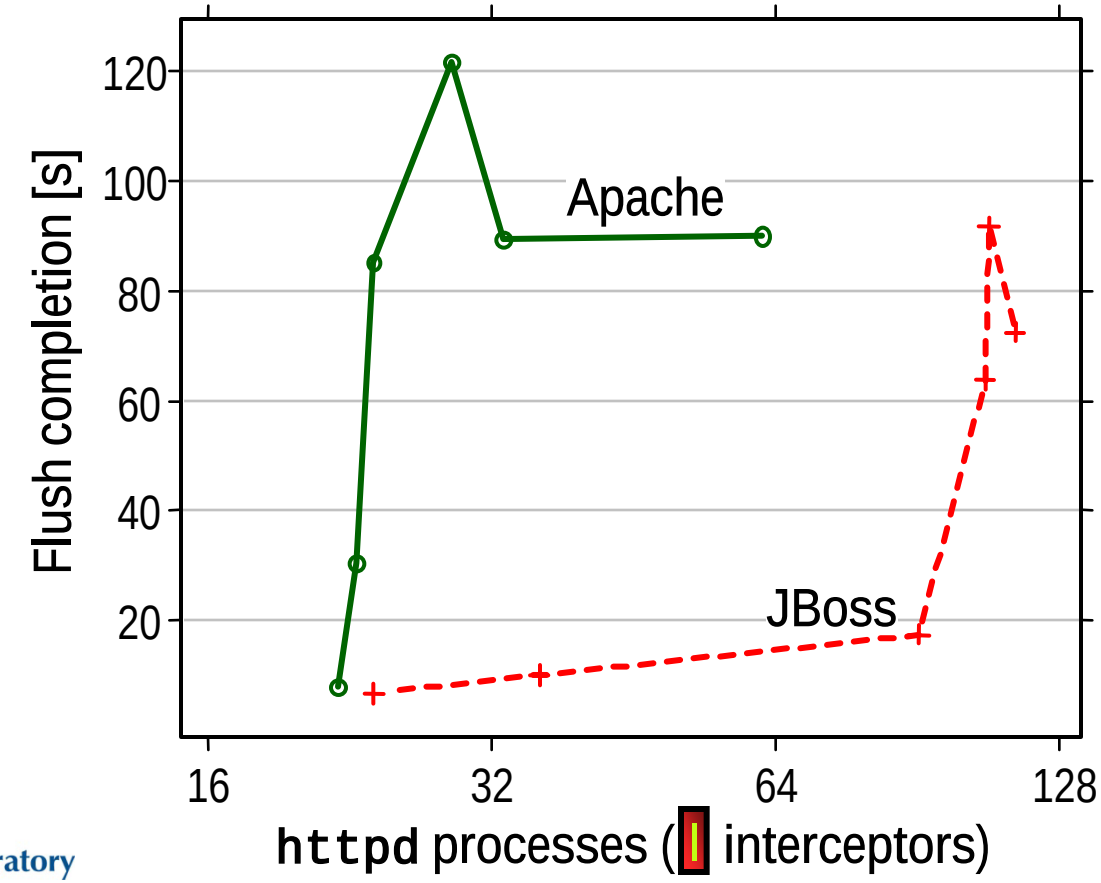




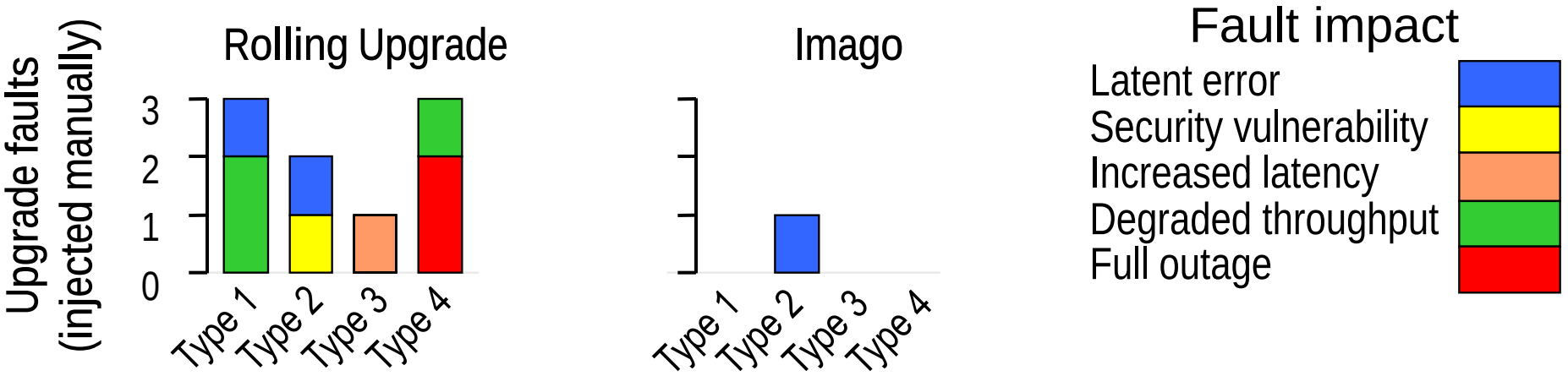
Imago: Quiescence Period

Evaluate the duration of the atomic switchover

- Need to flush in-progress requests



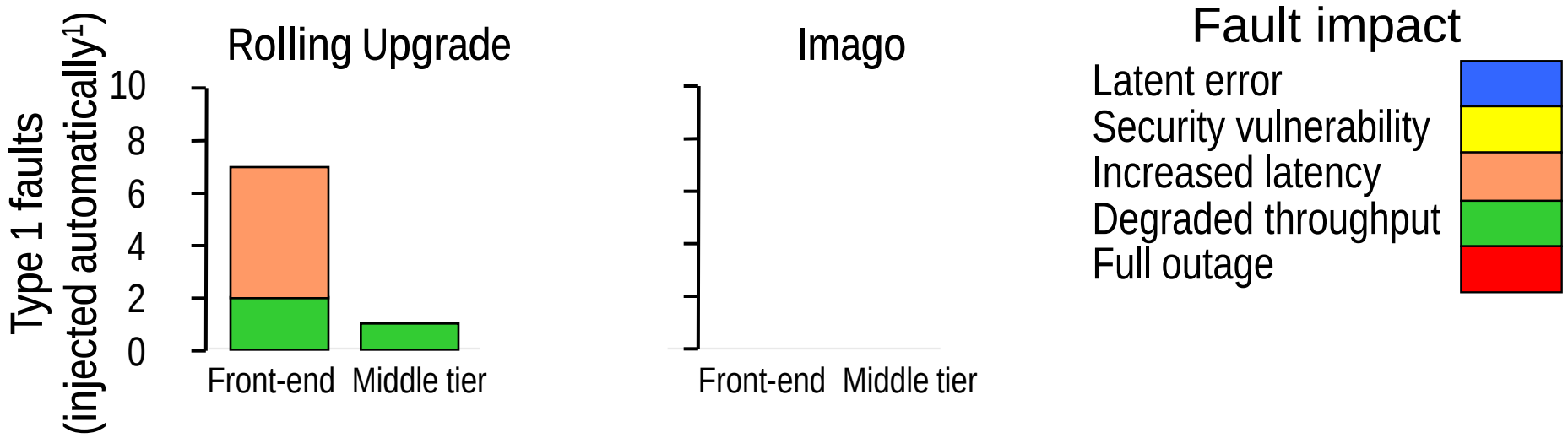
Upgrade-Fault Impacts



Upgrade-faults types

1. Simple configuration or procedural errors
2. Semantic configuration errors
3. Broken environmental dependencies
4. Data-access errors

Upgrade-Fault Impacts – cont'd



Upgrade-faults types

1. Simple configuration or procedural errors
2. Semantic configuration errors
3. Broken environmental dependencies
4. Data-access errors

[1] Using ConfErr [Keller+, 2008]

Summary of Findings

Isolation

- Prevents breaking *hidden dependencies*
  - Leading cause of upgrade failures
- Eliminates the *single points of failure* of in-place upgrades (e.g., the database for Type 4 faults)
- Avoids disrupting live workload

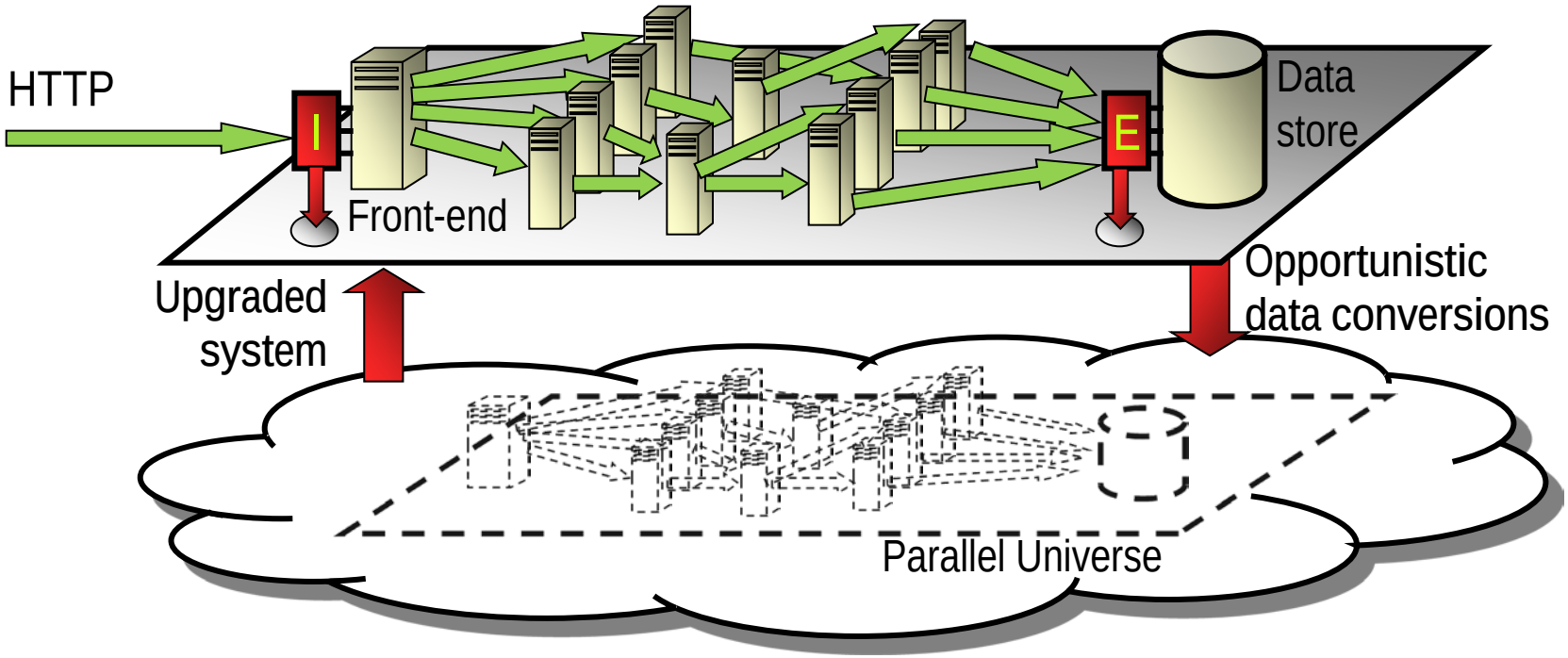
Atomicity

- Avoids states with mixed versions
- Prevents some *latent errors*

Fidelity

- Essential for online upgrades

Future Work: Upgrades-as-a-Service



- Lease resources *during* the upgrade
- Easier to use:
  - No dependency tracking
  - No mixed-version interactions

Conclusions

- Leading causes of upgrade failures
  - Breaking *hidden dependencies*
    - Inherent for in-place upgrades with mixed-version states
  - *Upgrade-centric fault model*: 4 common fault types
- Imago: dependable, online software upgrades
  - Isolation, atomicity, fidelity
  - *End-to-end upgrades* of distributed systems
- Enables *upgrades-as-a-service* model
  - More dependable
  - More automated