PerfBlower: A Novel Performance Testing Framework based on Virtual Amplification

Lu Fang, University of California, Irvine
Liang Dou, East China Normal University
Harry Xu, University of California, Irvine

2015-07-09

Performance Problems

- Inefficient code regions [G. Jin et al. PLDI 2012]

Performance Problems

- Inefficient code regions [G. Jin et al. PLDI 2012]

Simple Code Changes

Performance Problems

- Inefficient code regions [G. Jin et al. PLDI 2012]

Simple Code Changes

Performance Problems

- Inefficient code regions [G. Jin et al. PLDI 2012]

Simple Code Changes

- Widely exist
  - Firefox developers fix 50+/month over 10 years

Performance Problems

- Inefficient code regions [G. Jin et al. PLDI 2012]

Simple Code Changes

- Widely exist
  - Firefox developers fix 50+/month over 10 years
- Consequences
  - Financial loss, scalability reduction, etc.

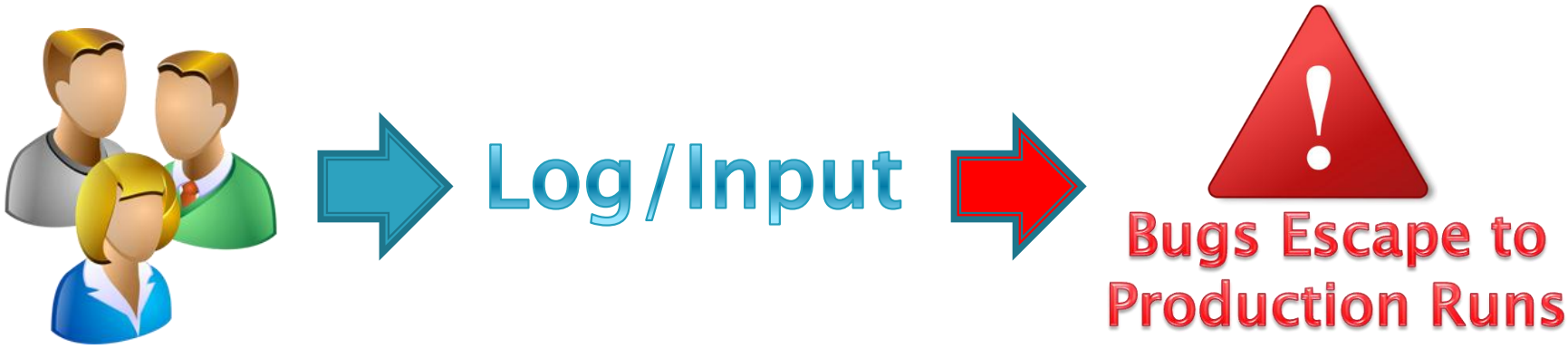
Motivation

- Existing solutions
  - Most are postmortem debugging techniques

Log/Input

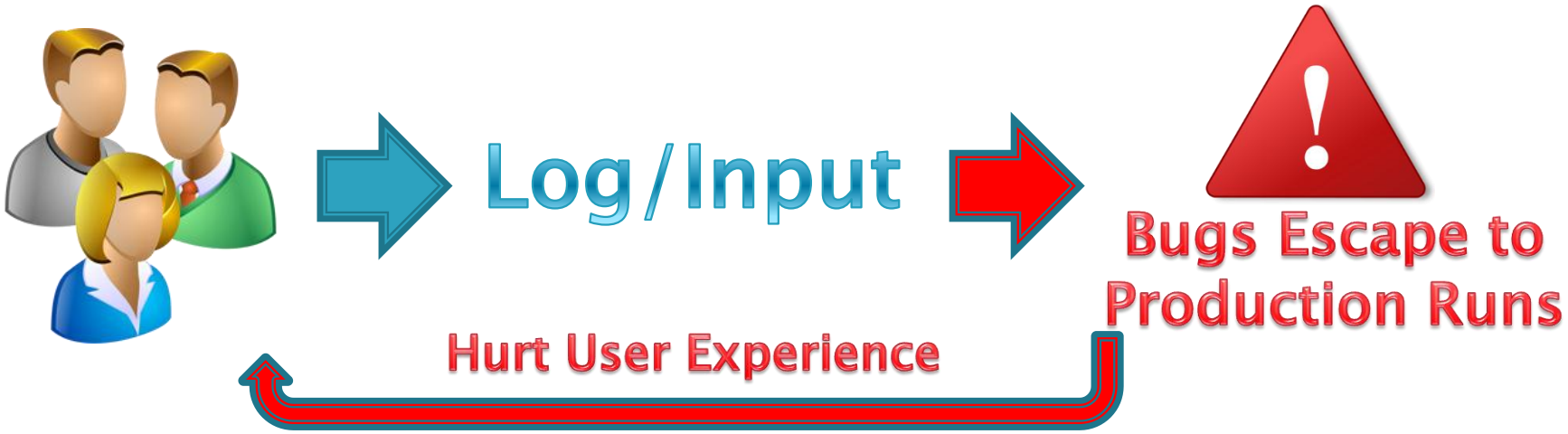
Motivation

- Existing solutions
  - Most are postmortem debugging techniques



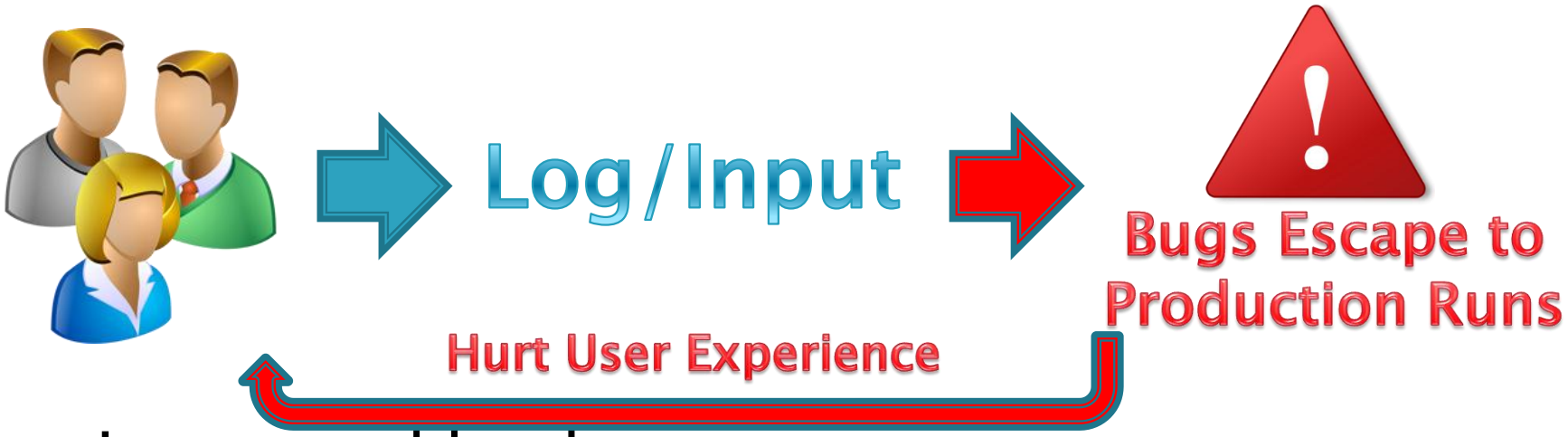
Motivation

- Existing solutions
  - Most are postmortem debugging techniques



Motivation

- Existing solutions
  - Most are postmortem debugging techniques



- Large workloads
  - To manifest performance problems
  - NOT available in testing environment, usually

Our Goal

- Find and fix performance problems
  - In the testing environment
  - Even without large workloads

Our Goal

- Find and fix performance problems
  - In the testing environment
  - Even without large workloads
- Focus on **memory related** performance problems
  - Such as memory leaks, inefficiently used containers, etc.

Our Goal

- Find and fix performance problems
  - In the testing environment
  - Even without large workloads
- Focus on **memory related** performance problems
  - Such as memory leaks, inefficiently used containers, etc.
  - Also **many redundant computations**
    - Such as unnecessary function calls

Our Solution – PerfBlower

- A novel performance testing framework
  - To detect memory related performance problems

Our Solution – PerfBlower

- A novel performance testing framework
  - To detect memory related performance problems
- General idea: amplify performance problems

Our Solution – PerfBlower

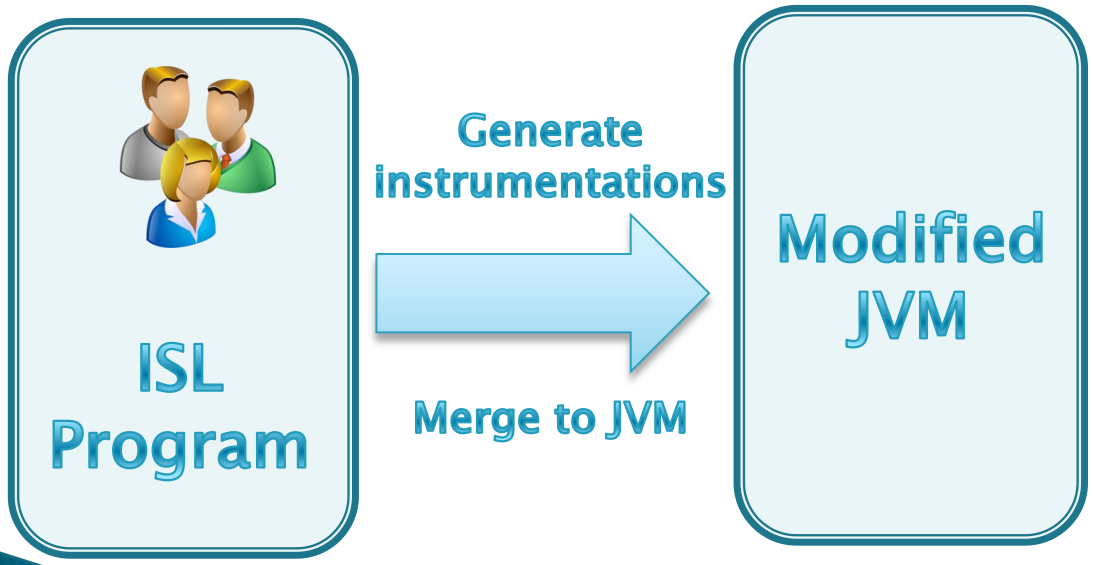
- A novel performance testing framework
  - To detect memory related performance problems
- General idea: amplify performance problems

ISL Program

Describe the symptoms and counter examples

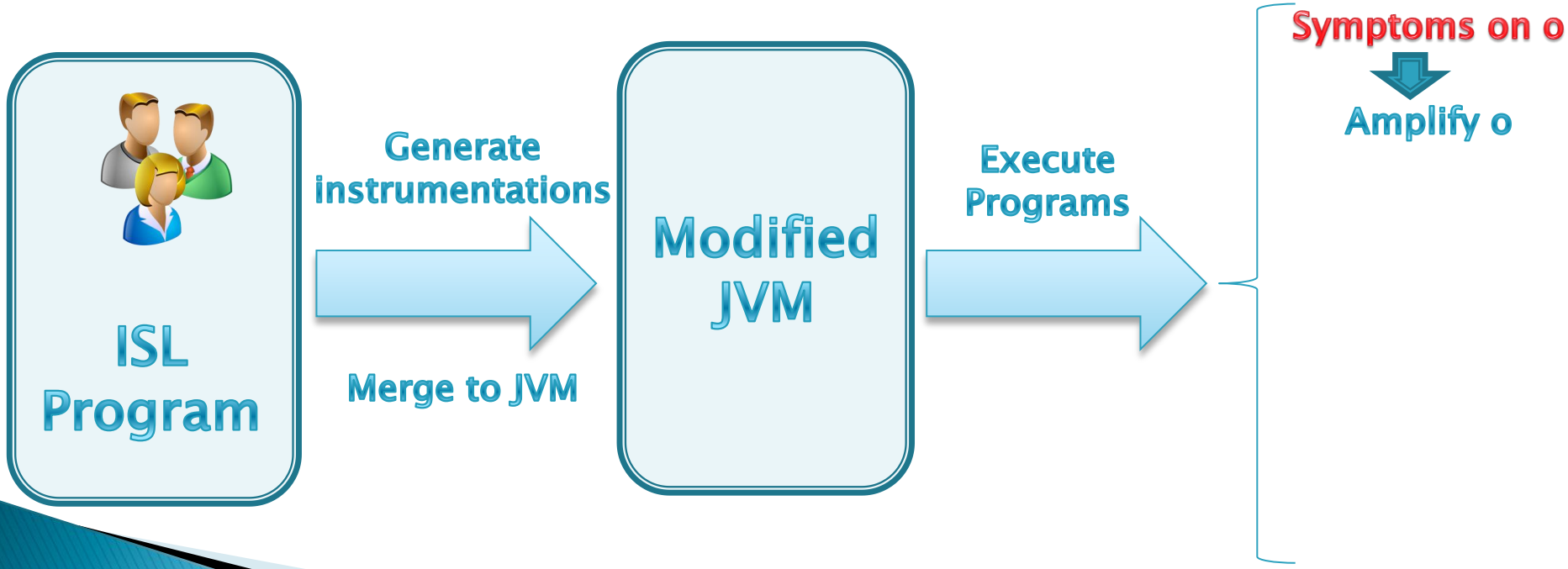
Our Solution – PerfBlower

- A novel performance testing framework
  - To detect memory related performance problems
- General idea: amplify performance problems



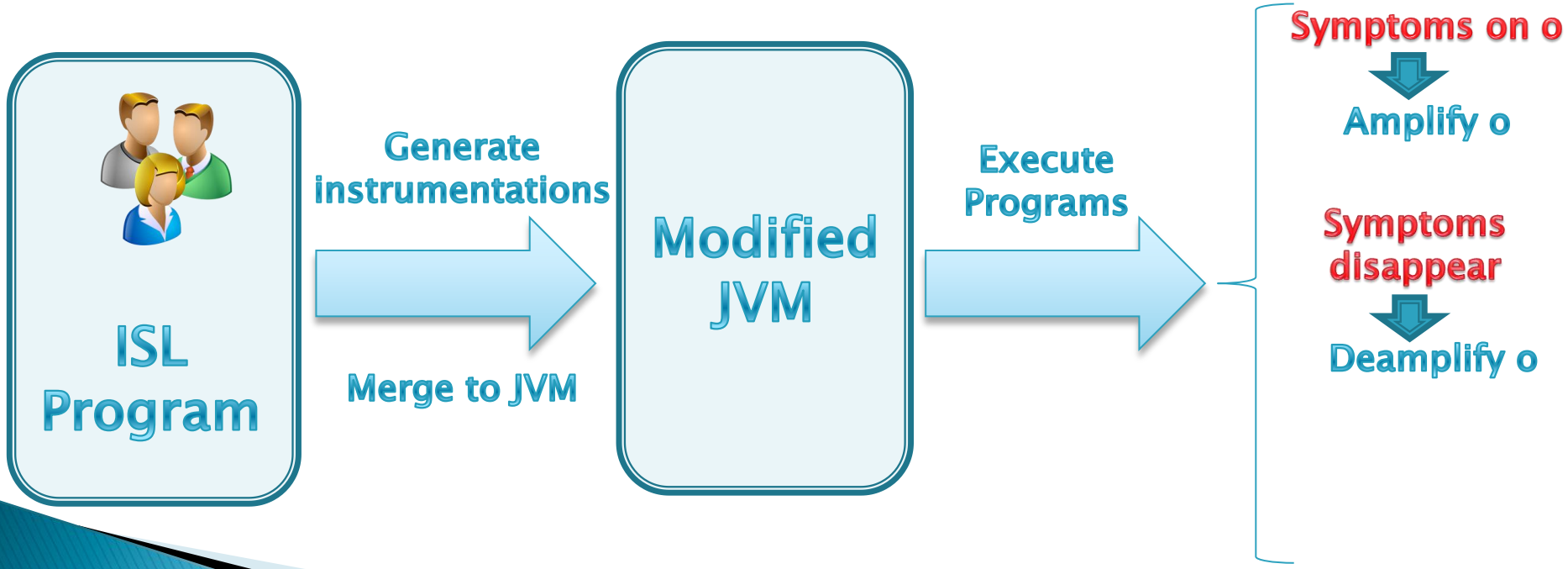
Our Solution – PerfBlower

- A novel performance testing framework
  - To detect memory related performance problems
- General idea: amplify performance problems



Our Solution – PerfBlower

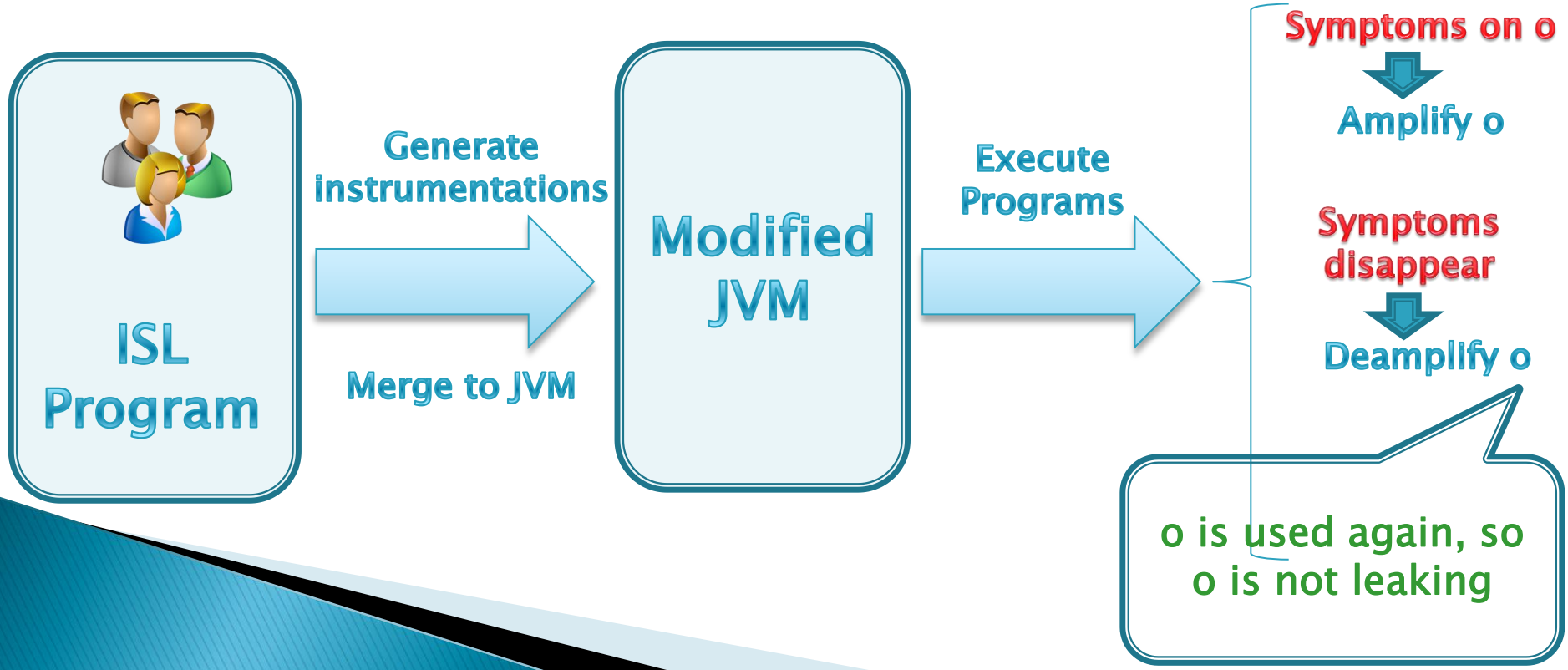
- A novel performance testing framework
  - To detect memory related performance problems
- General idea: amplify performance problems



Our Solution – PerfBlower

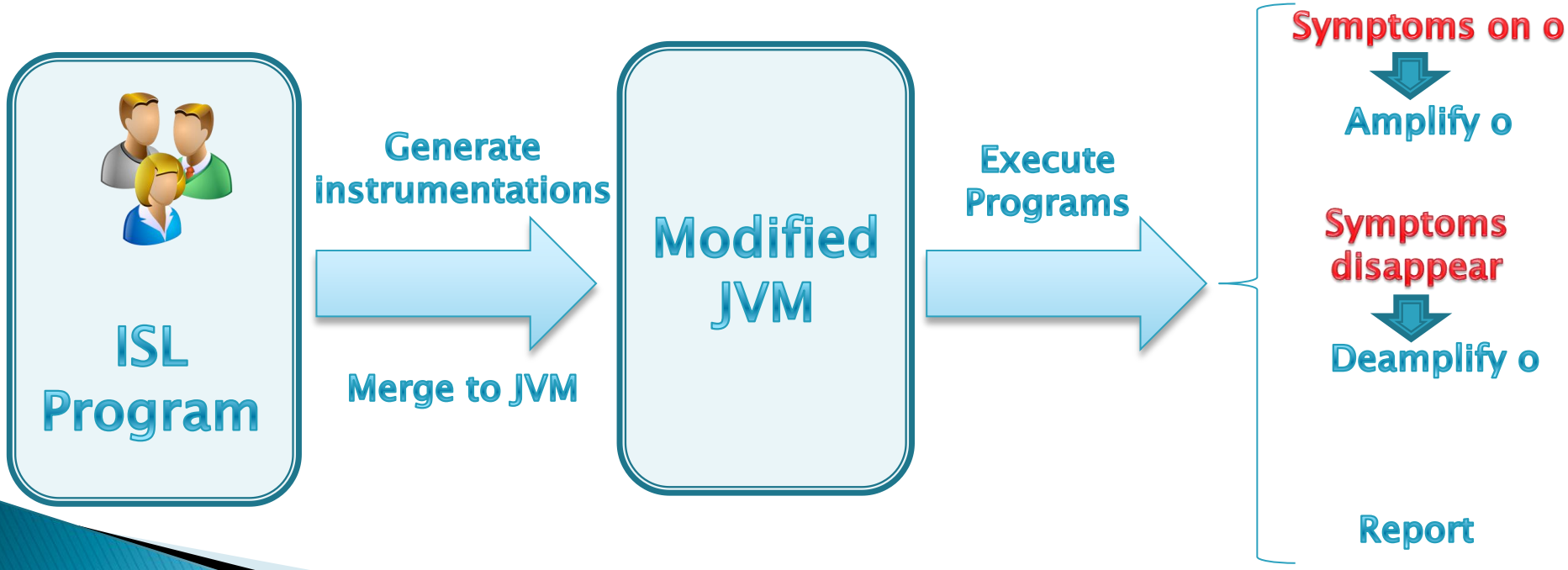
- A novel performance testing framework
  - To detect memory related performance problems
- General idea: amplify performance problems

e.g. to detect memory leaks, o is not used for a long time



Our Solution – PerfBlower

- A novel performance testing framework
  - To detect memory related performance problems
- General idea: amplify performance problems



Key Techniques

- Virtual amplification
  - Provide test oracles
- ISL (Instrumentation Specification Language)
  - Describe memory related performance problems
- Mirror chain
  - Record useful debugging information

Virtual Amplification

- Amplification (at object level)
  - Add space penalties to suspicious objects
  - Make the symptoms more obvious
  - Deamplification

Virtual Amplification

- Amplification (at object level)
  - Add space penalties to suspicious objects
  - Make the symptoms more obvious
  - Deamplification
- Virtual
  - Counter per object

Virtual Amplification

- Amplification (at object level)
  - Add space penalties to suspicious objects
  - Make the symptoms more obvious
  - Deamplification
- Virtual
  - Counter per object
- Virtual Space Overhead (VSO)
  - (P+S)/S
    - P is the sum of counters of all the tracked objects
    - S is the size of the live heap
  - Test oracle ➔ indicator of performance problems

Instrumentation Specification Language (ISL)

- Describe performance problems manifested by memory inefficiencies
  - such as memory leaks, etc.

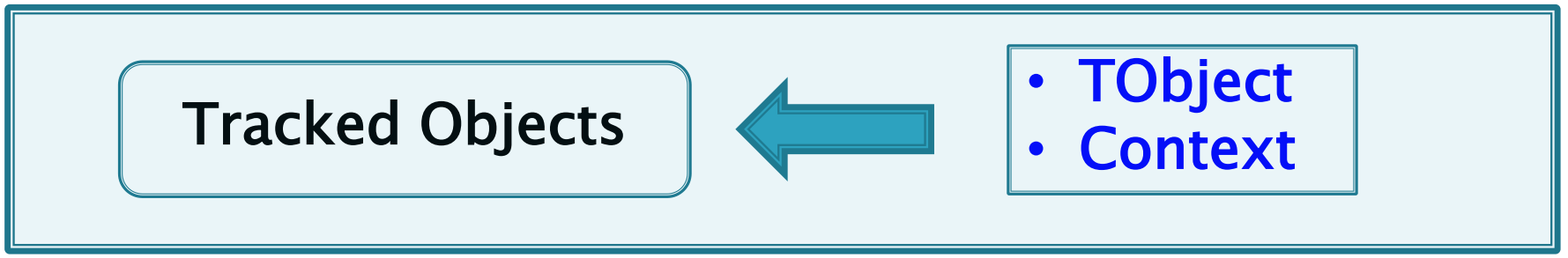
Instrumentation Specification Language (ISL)

- Describe performance problems manifested by memory inefficiencies
  - such as memory leaks, etc.
- A simple, event-based language
  - Describe symptoms/counterexamples
  - Specify the corresponding actions

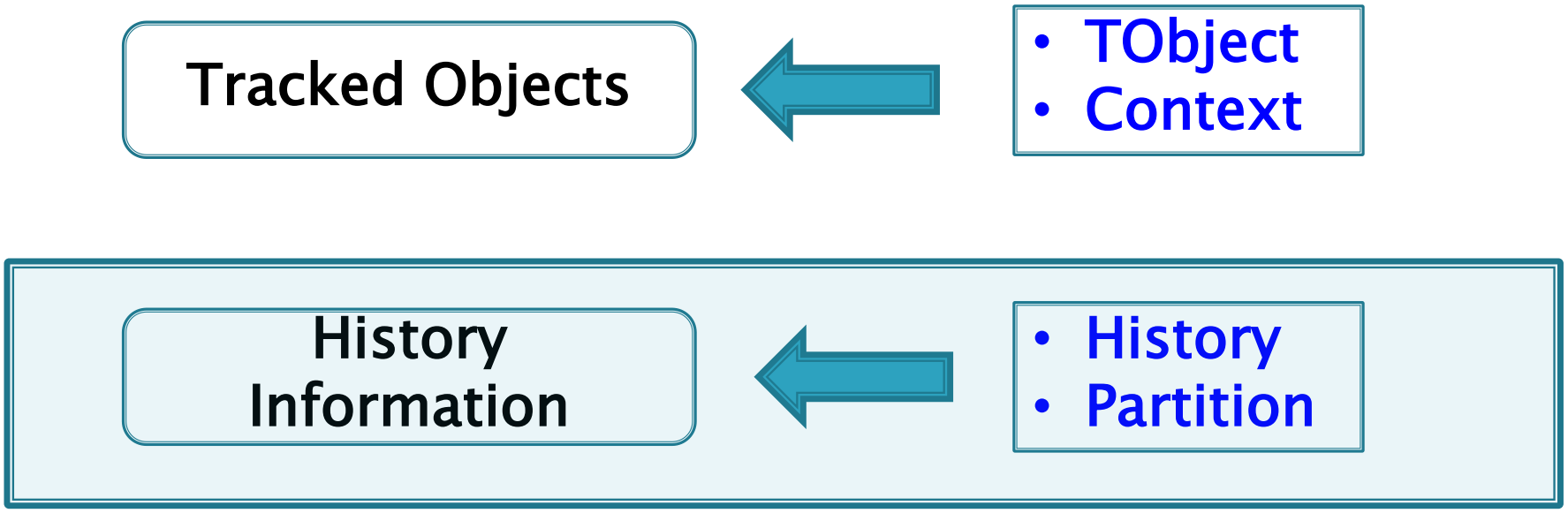
Instrumentation Specification Language (ISL)

- Describe performance problems manifested by memory inefficiencies
  - such as memory leaks, etc.
- A simple, event-based language
  - Describe symptoms/counterexamples
  - Specify the corresponding actions
- An ISL program consists of:
  - TObject
  - Context
  - History
  - Partition
  - Event

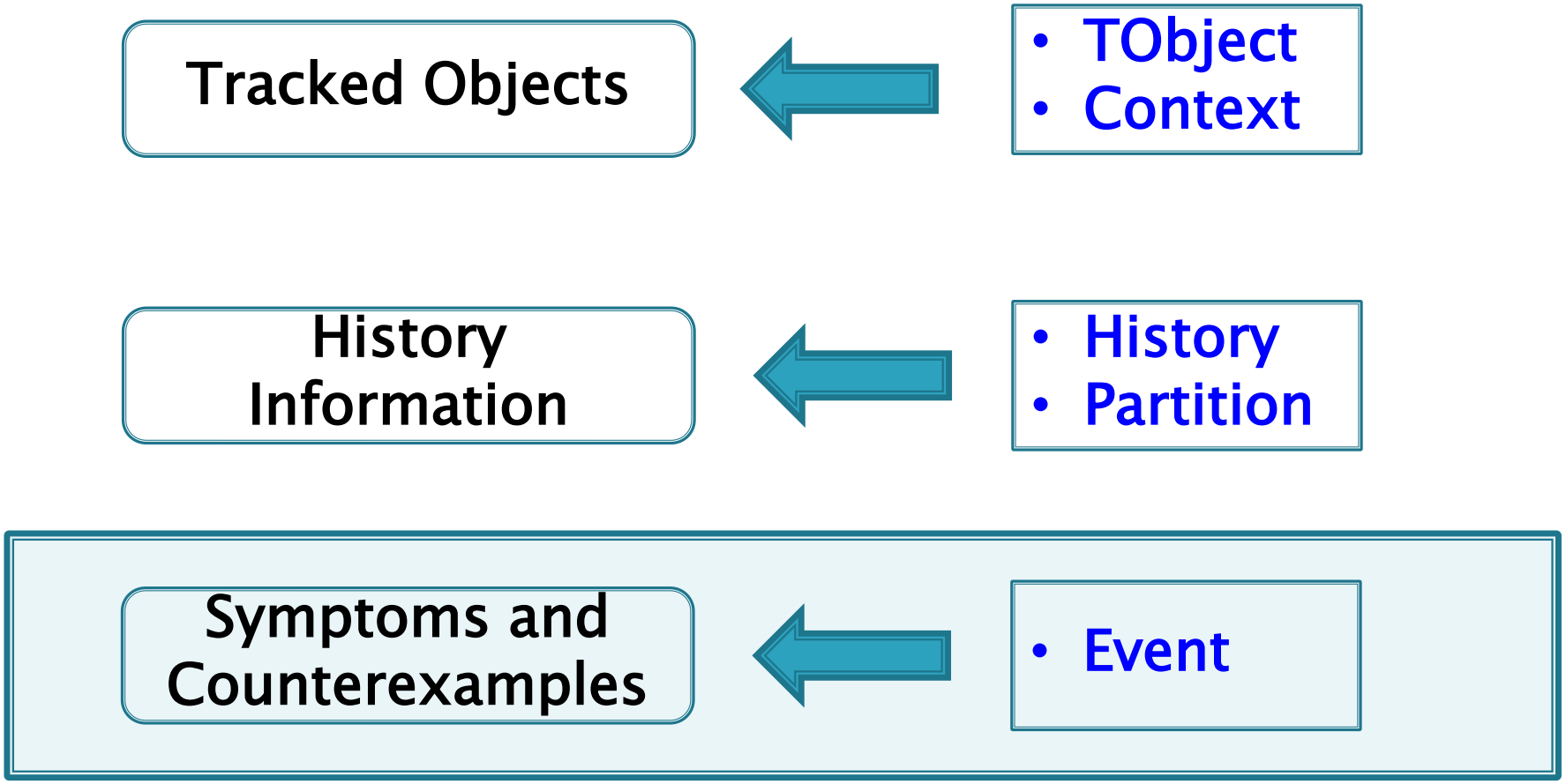
How to Use ISL?



How to Use ISL?



How to Use ISL?



An ISL Program for Memory Leak

- Memory leaks in Java programs
  - **Useless** objects cannot be reclaimed because of **unnecessary** references
- Memory leak symptoms in Java programs
  - Leaking objects are neither read nor written any more (**stale**)
- The counterexamples of memory leaks
  - The object is accessed again

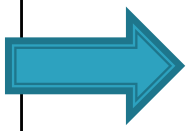
An ISL Program for Memory Leak

```
Context TrackingContext {
    sequence = "*,*.main,*";
    type = "java.lang.Object";
}

TObject MyObj{
    include = TrackingContext;
    partition =  P;
    instance boolean useFlag =
        false; //Instance Field
}

History UseHistory {
    type = "boolean";
    size = 10;//User Parameter
}

Partition P {
    kind = all;
    history = UseHistory;
}
```



Specify the tracked objects

Context : (1) Calling Context
(2) Object Type

TObject : Tracked Object Specification

```
Event on_rw(Object o, Field f,
            Word w1, Word w2){
    o.useFlag = true;
    deamplify(o);
}

Event on_reachedOnce(Object o){
    UseHistory h = getHistory(o);
    h.update(o.useFlag);
    if(h.isFull()
        && !h.contains(true)){
        amplify(o);
    }
}
```

An ISL Program for Memory Leak

```
Context TrackingContext {
     sequence = "*,*.main,*";
     type = "java.lang.Object";
}

TObject MyObj{
     include = TrackingContext;
     partition =  P;
     instance boolean useFlag =
          false; //Instance Field
}

History UseHistory {
     type = "boolean";
     size = 10;//User Parameter
}

Partition P {
     kind = all;
     history = UseHistory;
}
```

Record history information

History : Execution Windows

Partition: (1) Heap Partitioning

(2) The Binding History

```
Event on_rw(Object o, Field f,
            Word w1, Word w2){
     o.useFlag = true;
     deamplify(o);
}

Event on_reachedOnce(Object o){
     UseHistory h = getHistory(o);
     h.update(o.useFlag);
     if(h.isFull()
          && !h.contains(true)){
          amplify(o);
     }
}
```

An ISL Program for Memory Leak

```
Context TrackingContext {
    sequence = "*,*.main,*";
    type = "java.lang.Object";
}

TObject MyObj{
    include = TrackingContext;
    partition =  P;
    instance boolean useFlag =
        false; //Instance Field
}

History UseHistory {
    type = "boolean";
    size = 10;//User Parameter
}

Partition P {
    kind = all;
    history = UseHistory;
}
```

Define the actions

Event: (1) Counterexamples
(2) Symptoms

```
Event on_rw(Object o, Field f,
            Word w1, Word w2){
    o.useFlag = true;
    deamplify(o);
}

Event on_reachedOnce(Object o){
    UseHistory h = getHistory(o);
    h.update(o.useFlag);
    if(h.isFull()
        && !h.contains(true)){
        amplify(o);
    }
}
```

Reference Path

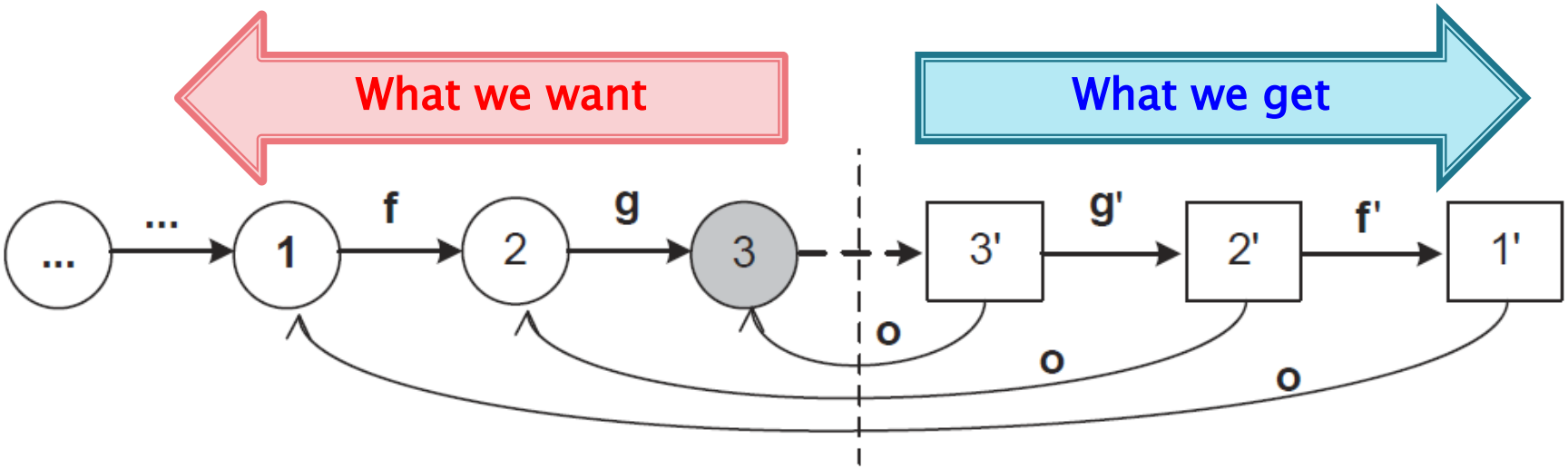
- Very useful for debugging [G. Xu et al, PLDI 2011]
  - Reveal both calling context and data structures

Reference Path

- Very useful for debugging [G. Xu et al, PLDI 2011]
  - Reveal both calling context and data structures
- Difficult to obtain
  - Backward information of object graph is not available
  - In practice, GC implementations use BFS

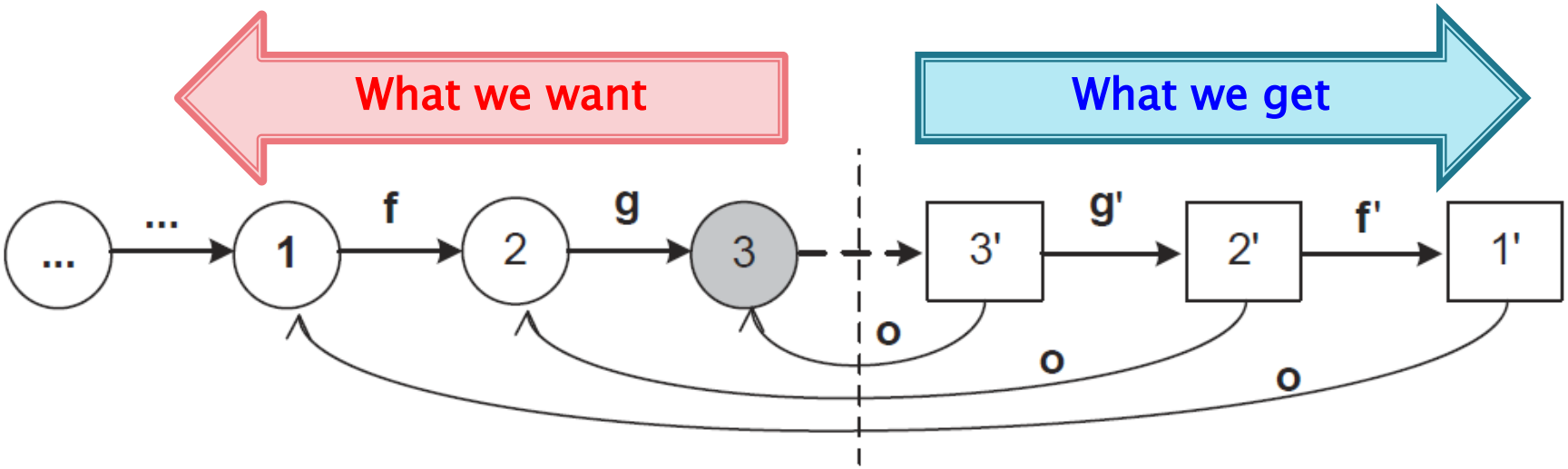
Mirror Chain

- A mirror of reference path



Mirror Chain

- A mirror of reference path

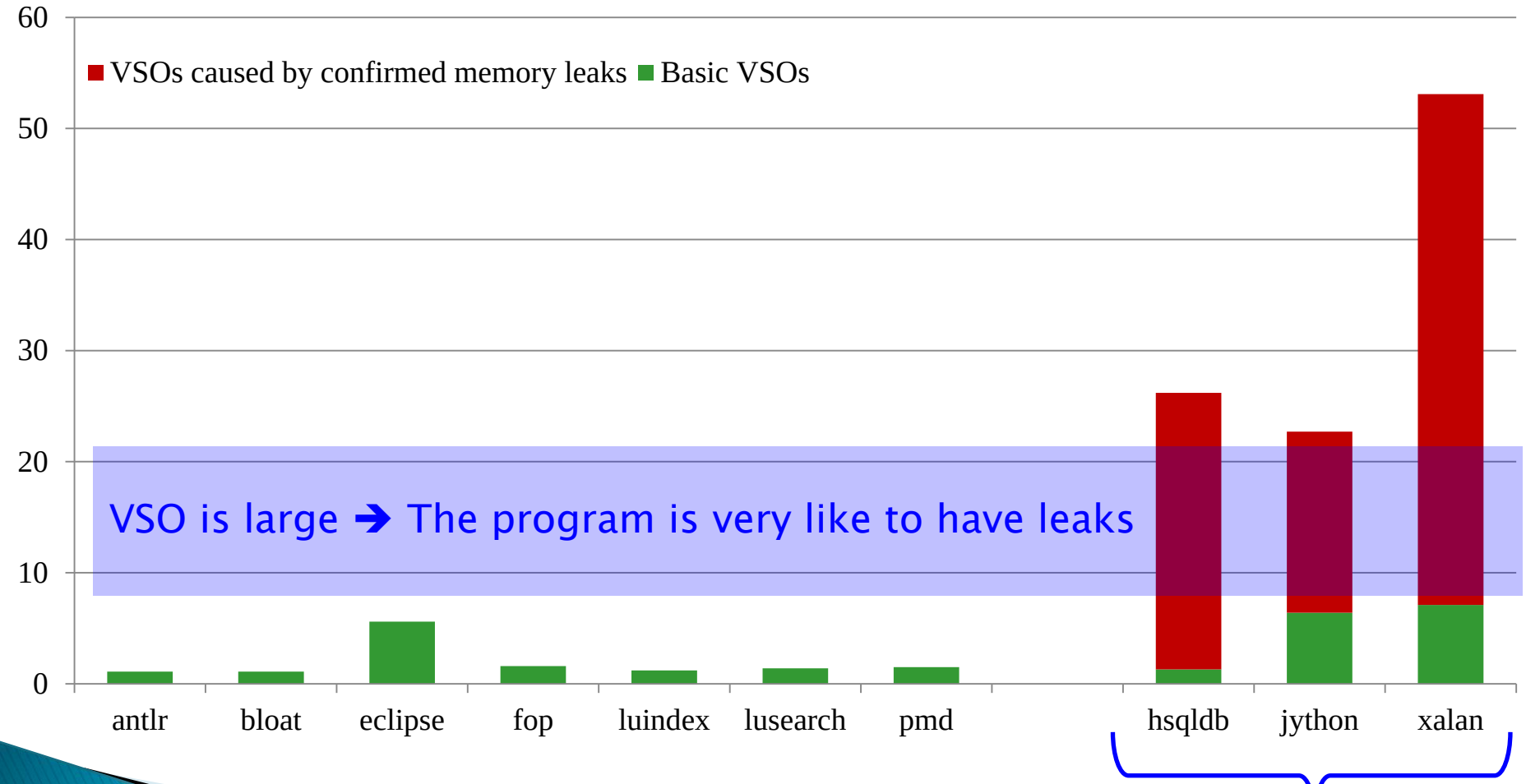


- An algorithm to build the mirror chain
  - Details can be found in our paper

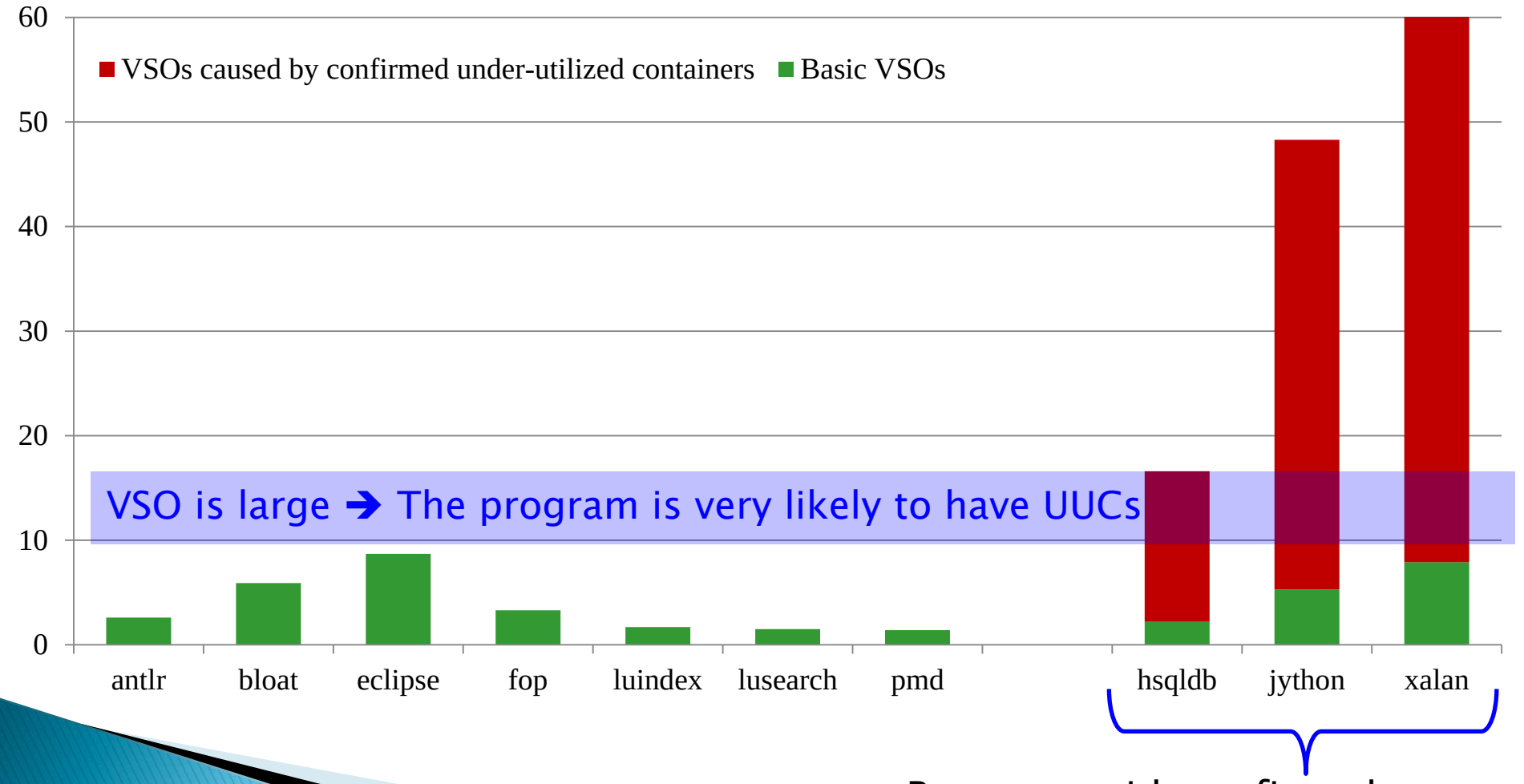
Evaluations

- We have implemented 3 amplifiers
  - Memory leak amplifier
  - Under-utilized container (UUC) amplifier
  - Over-populated container (OPC) amplifier
- Benchmarks
  - DaCapo benchmark suite [S. Balckburn, et al. OOPSLA 2006]
- Totally we have found 11 performance problems
  - 8 unknown problems
  - 3 known problems

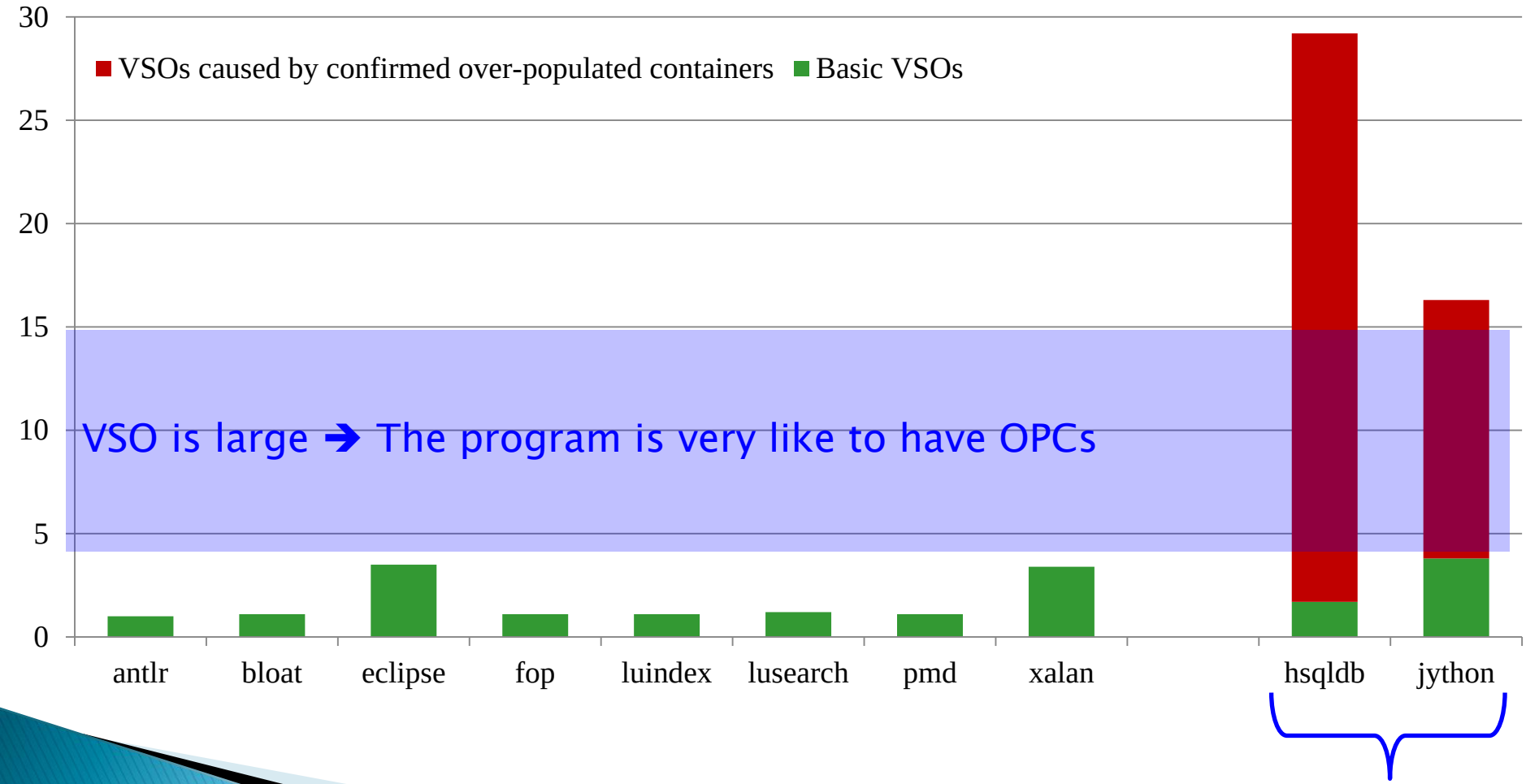
VSOs Reported by Memory Leak Amplifier



VSOs Reported by Under-Utilized Container Amplifier



VSOs Reported by Over-Populated Container Amplifier



Programs with confirmed unknown OPCs

Performance Improvements

Benchmark	Space Reduction	Time Reduction
xalan-leak	25.4%	14.6%
jython-leak	24.3%	7.4%
hsqldb-leak	15.6%	3.1%
xalan-UUC	5.4%	34.1%
jython-UUC	19.1%	1.1%
hsqldb-UUC	17.4%	0.7%
hsqldb-OPC	14.9%	2.9%

Runtime Overheads

- Time overheads
  - Memory leak amplifier: 2.39x
  - Under-utilized container: 2.74x
  - Over-populated container: 2.73x
- Space overheads
  - Memory leak amplifier: 1.23x
  - Under-utilized container: 1.23x
  - Over-populated container: 1.25x

Conclusions

- Propose a framework for performance testing
- Develop compiler and runtime system support
- Successfully amplify three different types of performance problems
  - Help developers find and fix performance problems even in the testing environment

Thanks!

Q&A