

Preserving Reactiveness

Understanding and Improving the Debugging Practice of Blocking-Call Bugs

Arooba Shahoor

School of Computer Science and Engineering
Kyungpook National University
Daegu, South Korea

Dongsun Kim

Associate Professor
Department of Computer Science and Engineering
Korea University
Seoul, South Korea

Jooyong Yi

Associate Professor
Dept. of Computer Science and Engineering
UNIST
Ulsan, South Korea



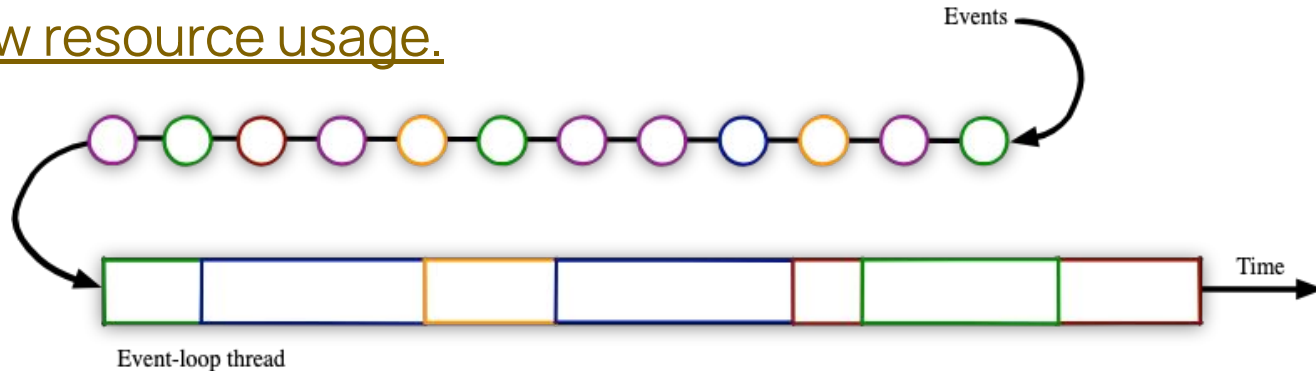
ISSTA 2024, 19th September 2024

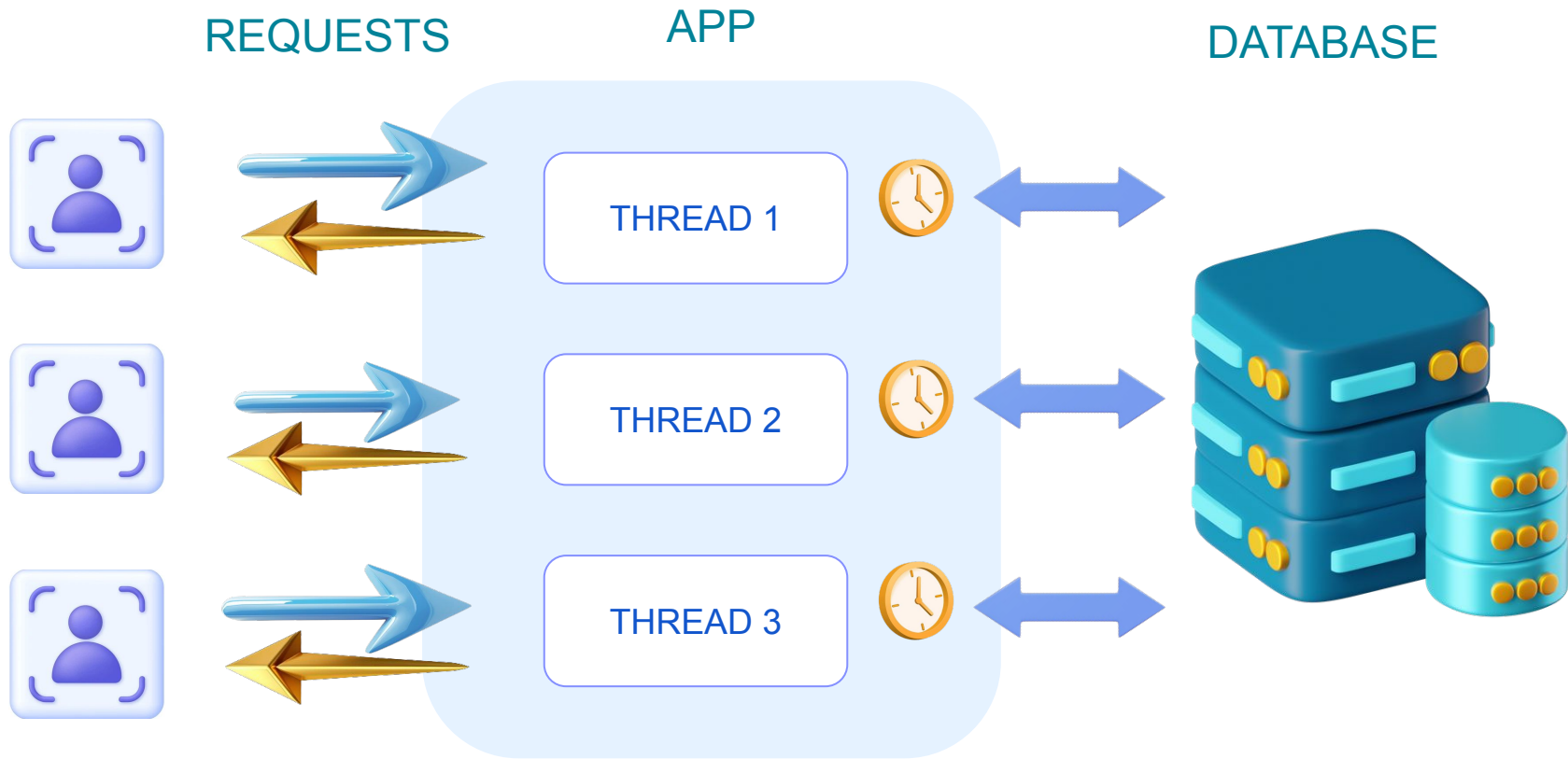
Why Reactive Programming?

What problems does it solve

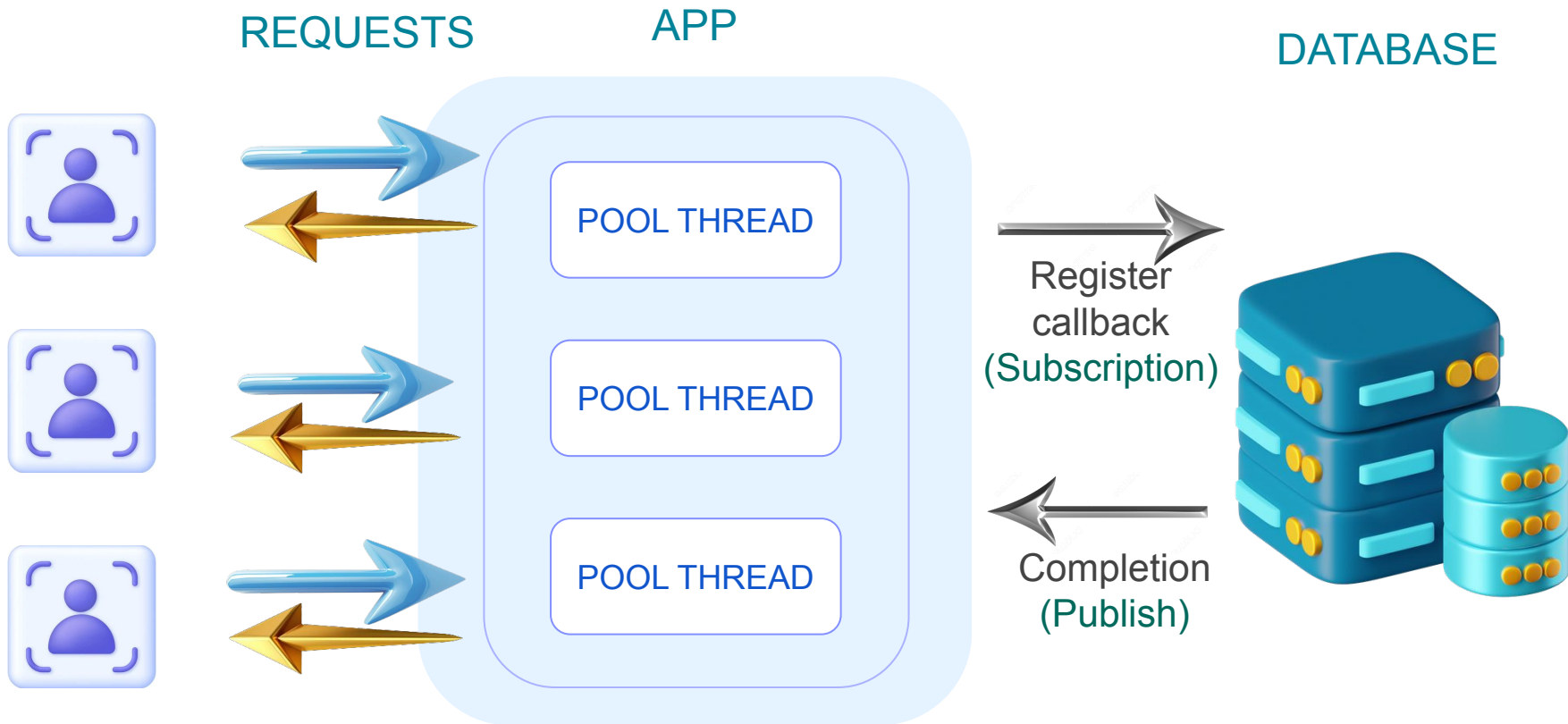
Reactive Programming = Efficient Use of Threads

- ❑ Turns synchronous operations into asynchronous **events streams**.
- ❑ Single, non-blocking thread a.k.a **EventLoop** thread.
- ❑ *Callback* notifies once blocking task completes.
- ❑ Ideal for rapidly scaling apps.
- ❑ Apps requiring low resource usage.





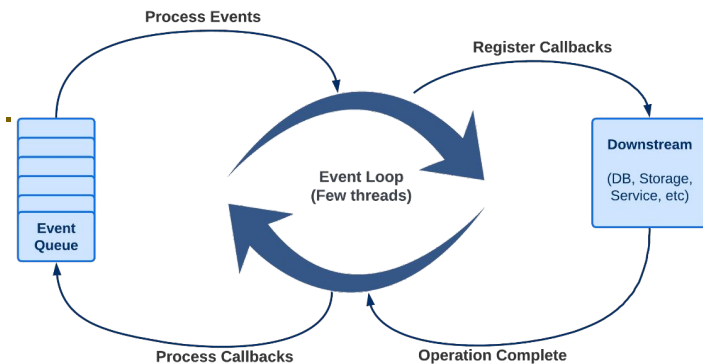
Synchronous model



Reactive model

Event Loop: Efficient Use of Threads

- ❑ Maintains a small, fixed-size thread pool.
- ❑ Request returns a **publisher** that can be **subscribed** to.
- ❑ Publisher and subscriber can operate in *different* threads.
- ❑ Thread not blocked while data is fetched.
- ❑ Thread available to handle other requests.



Pitfalls to watch out for

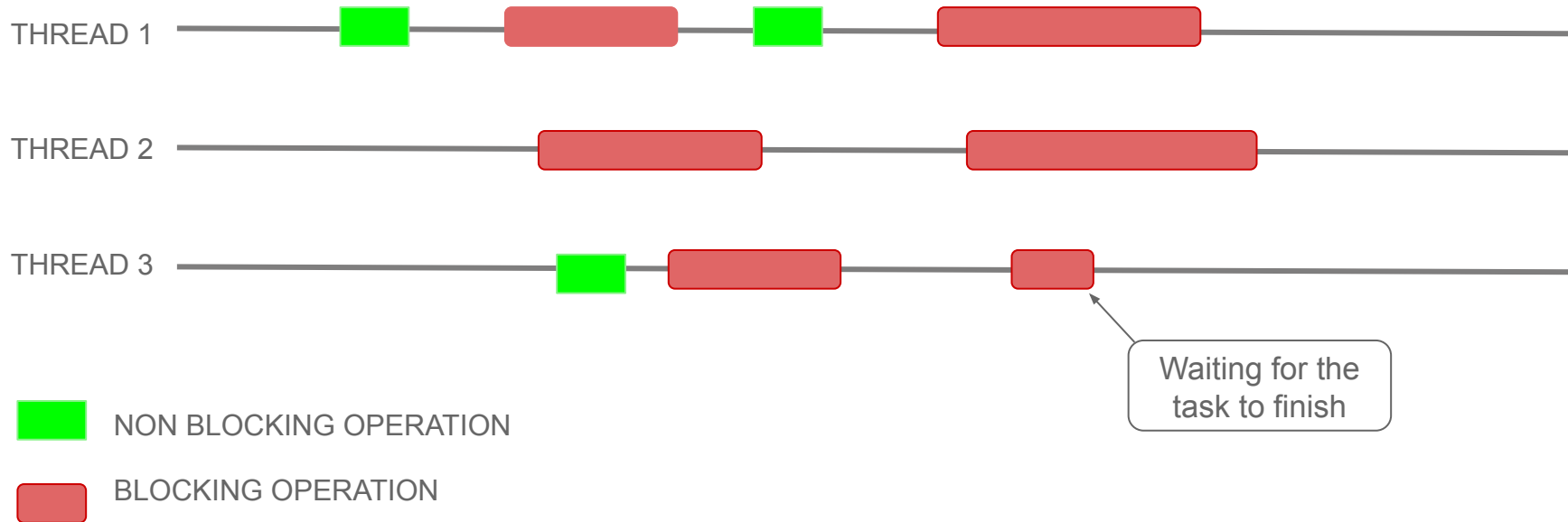
—

Being **end to end** reactive is
the key

- Thorough understanding of the **pub-sub** design pattern
- Appropriate usage of reactive operators

Blocking Calls in Reactive Programming

“Any operation that pauses a thread until it completes or a condition is met.”



Impact of Blocking Calls in Reactive Apps

Blocking even a **single** request impacts the latency of **all**.



Impact of Blocking Calls in Reactive Apps

- Thread pool exhausted very soon.
- Freezing of the entire event loop.
- Unresponsiveness; No new events can be processed.
- Unexpected behavior caused by delays.
- Scalability and concurrency severely compromised.

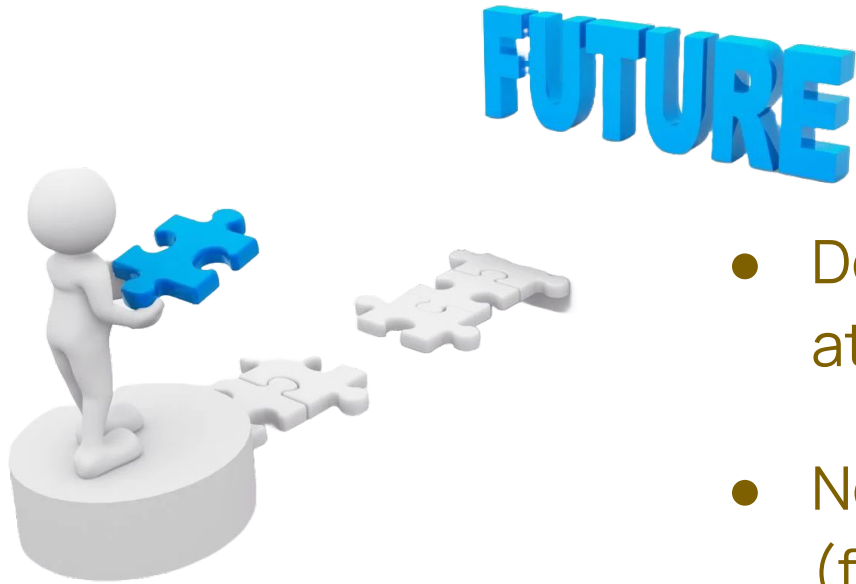


Why Blocking Call Bugs Are Not Given Attention

- Hard to detect.
- Hard to reproduce.
- Require large refactoring.



Proactive Program Repair



- Does not require test cases at the beginning.
- Non-intrusive fixes (functionality-preserving).
- Replication becomes supplementary.

What We Are After

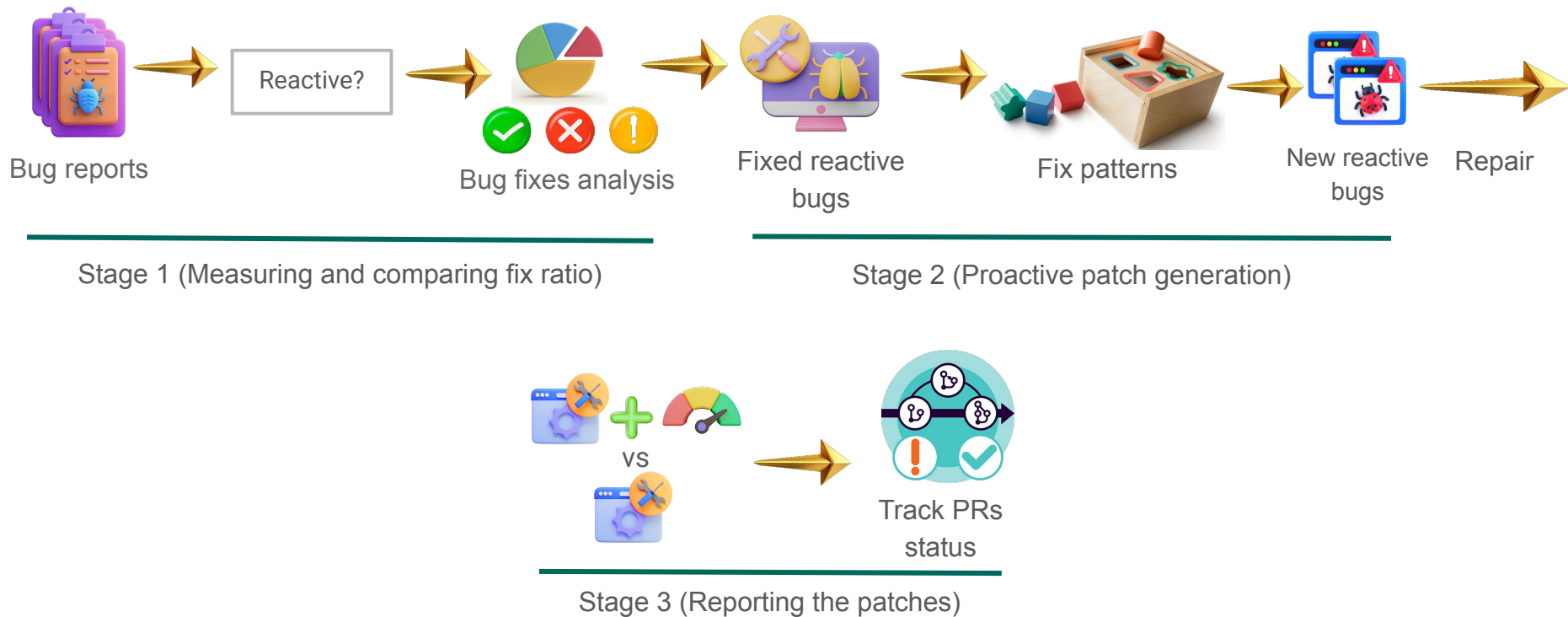
RQ1: How different is the **fix ratio** between reactiveness bugs and non-reactiveness bugs?

RQ2: Does our proactive repair technique generate **non-intrusive** patches fixing blocking-call bugs?

RQ3: Are the patches **more acceptable** if they are submitted with improvement evidence?



Approach



Performance Metrics

Scalability and Resource efficiency

- Latency
- CPU
- Memory (heap and RAM)



5 Common Fix Patterns

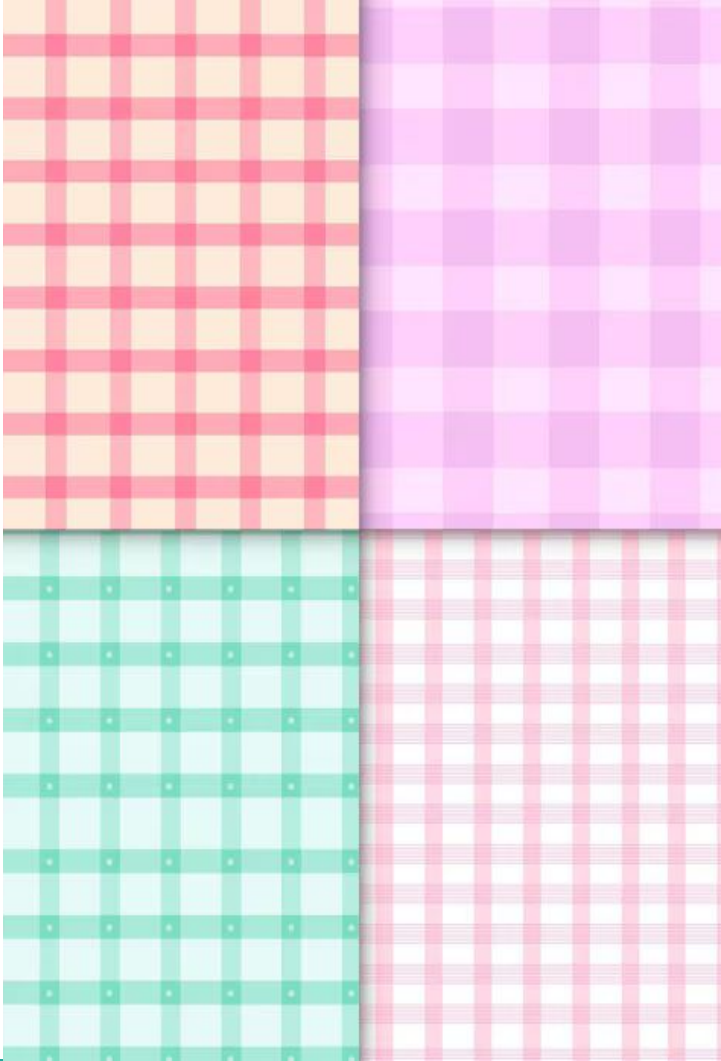
FP1. Offloading to Separate Worker Threads

FP2. Lazy Method Call

FP3. Reactive Filtering

FP4. Non-blocking Chaining

FP5. Non-blocking Subscription



FP1. Offloading to Separate Worker Threads

Blocking Call Pattern

```
private Mono<T> methodWithReactiveReturnType(args..) {  
    ...  
    return blockingOperation();  
}
```

Fix Pattern

```
private Mono<T> methodWithReactiveReturnType(args..) {  
    ...  
    return blockingOperation()  
        .subscribeOn(Schedulers.elastic());  
}
```

Fix Ratio Analysis (RQ 1)



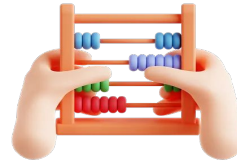
Collect reactive projects



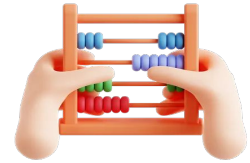
Verify project criteria



Classify bugs by reactivity



Count fixed vs. unfixed **reactive** bugs

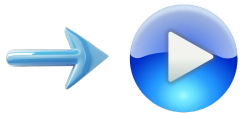


Count fixed vs. unfixed **non-reactive** bugs

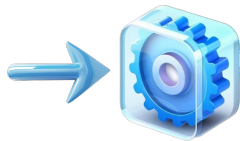
Effectiveness Analysis (RQ 2)



Subject with
blocking call



Run the
subject



Execute test suite



Note performance



Apply fix



Run the subject



Execute test suite



Note **performance**



Compare before and after
**test cases results and
performance**

```
DEBUG CONSOLE  PROBLEMS  OUTPUT  TERMINAL

[PASS] src/components/Items.test.js
  ✓ renders correctly when there are no items (8ms)

> 1 snapshot written.
Snapshot Summary
  > 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  1 written, 1 total
Time:        0.889s, estimated 1s
Ran all test suites related to changed files.

Watch Usage
  > Press a to run all tests.
  > Press f to run only failed tests.
  > Press q to quit watch mode.
  > Press p to filter by a filename regex pattern.
  > Press t to filter by a test name regex pattern.
  > Press Enter to trigger a test run.
```



```
DEBUG CONSOLE  PROBLEMS  OUTPUT  TERMINAL

[PASS] src/components/Items.test.js
  ✓ renders correctly when there are no items (8ms)

> 1 snapshot written.
Snapshot Summary
  > 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  1 written, 1 total
Time:        0.889s, estimated 1s
Ran all test suites related to changed files.

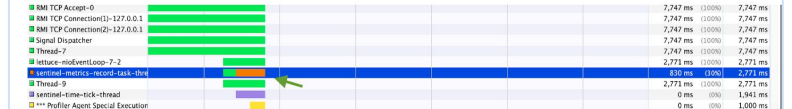
Watch Usage
  > Press a to run all tests.
  > Press f to run only failed tests.
  > Press q to quit watch mode.
  > Press p to filter by a filename regex pattern.
  > Press t to filter by a test name regex pattern.
  > Press Enter to trigger a test run.
```

Before

(The main thread only sits idle in waiting and parked state)



After



Acceptance Analysis (RQ 3)



Compare *before* and *after* performance

Is performance improved?

Yes

Is unknown blocking call?

Yes



Submit fix PR **with** performance results



Submit fix PR **without** performance results



Track PR status

Fix blocking calls #8

New issue

Before

(The main thread only sits idle in waiting and parked state)



Apparently the chat module has some blocking calls (as detected by BlockHound):

```
> reactor.blockhound.BlockingOperationError: Blocking call! java.lang.ProcessImpl.waitForProcessExit()
Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Error has been observed at the following site(s):
* _checkpoint - HTTP POST /message [ExceptionHandler]
Original Stack Trace:
at java.base/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:233) ~[java:11]
at java.base/java.util.concurrent.locks.LockSupport.park(LockSupport.java:161) ~[java:11]
at java.base/java.util.concurrent.ForkJoinPool.managedBlock(ForkJoinPool.java:363) ~[java:11]
at java.base/java.util.concurrent.ForkJoinPool.managedBlock(ForkJoinPool.java:363) ~[java:11]
at java.base/java.util.concurrent.ForkJoinPool.managedBlock(ForkJoinPool.java:363) ~[java:11]
```

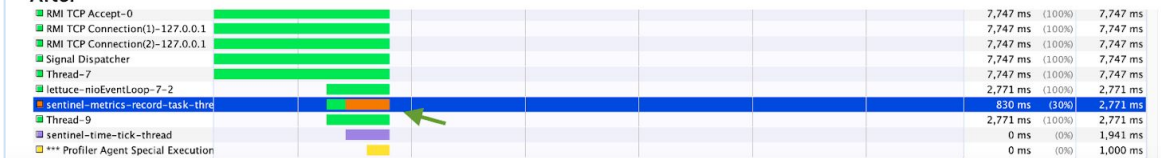
Assignees

No one assigned

Labels

None yet

After



RESULTS



RQ1: How different is the **fix ratio** between reactiveness bugs and non-reactiveness bugs?

- ❑ Fixed non-reactiveness bugs: **88.6%**
- ❑ Fixed reactiveness bugs: **60.9%**



- ❑ Blocking-call bugs are one of the most common unfixed and ignored bugs.

RQ2: Does our proactive repair technique generate **non-intrusive** patches fixing blocking-call bugs?

- ❑ Fixed 30 blocking-call bugs.
- ❑ Verified via performance improvement.
- ❑ Non-intrusive (preserved original functionality).
- ❑ Test cases, program execution and developers response.



RQ3: Are the patches more acceptable if they are submitted with improvement evidence

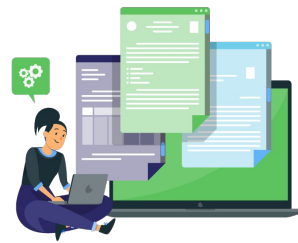
- ❑ **8** out of 15 PRs (**53%**) with performance analysis results are accepted.
- ❑ **3** out of 15 (**20%**) PRs are accepted without the results.





Practitioners Recommendations

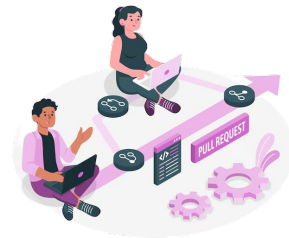
Improve Documentation: Threading and blocking operations.



Enhance Training: Async & non-blocking skills.



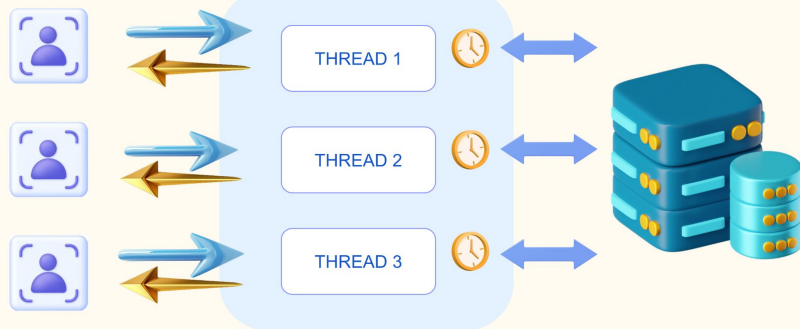
Keep PRs short & sweet, but with evidence!



REQUESTS

APP

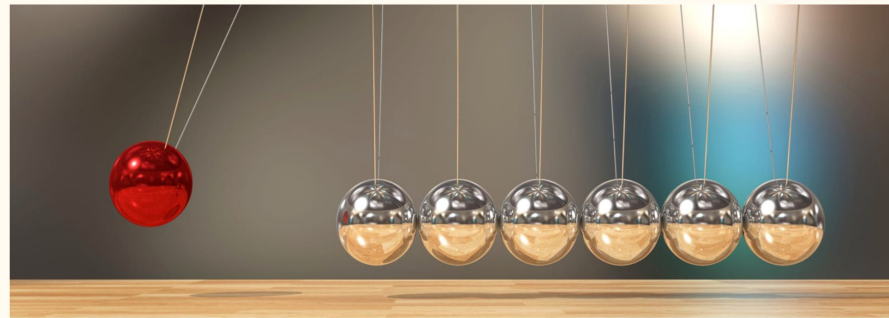
DATABASE



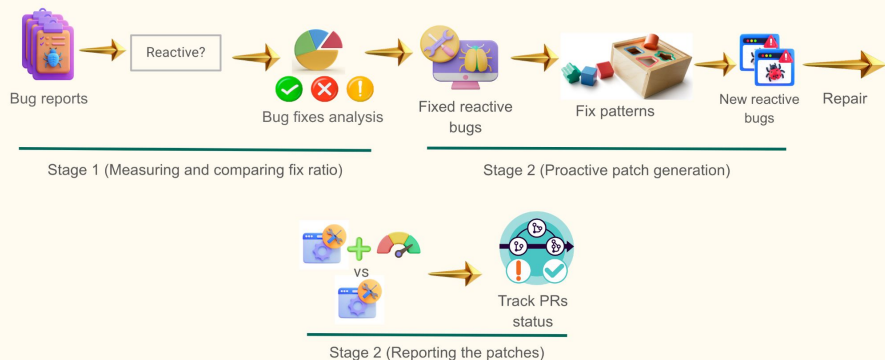
Synchronous model

Impact of Blocking Calls

Blocking even a **single** request impacts the latency of **all**.



Approach



RQ1: How different is the **fix ratio** between reactivity bugs and non-reactivity bugs?

Fixed non-reactiveness bugs: **88.6%**

Fixed reactivity bugs: **60.9%**



Blocking-call bugs are one of the most common unfixed and ignored bugs

Other Programming Languages

- Async/await, coroutines, asynchronous libraries.
- Similar blocking call bugs and fixes shared by **Kotlin**, **Python** and **Rust**.

