Lancaster University Leipzig

# Benchmarking Change Detection Exactness and Overhead of Instrumentation and Sampling

**David Georg Reichelt**[1,2], Juozas Skarbalius[1]

[1]School of Computing and Communications, Lancaster University Leipzig

[2]University Computing Centre, Universität Leipzig

12.5.2025

# Main Use Case of Observability: Change Detection

- Observability requires telemetry: Instrumentation or sampling

# Main Use Case of Observability: Change Detection

- Observability requires telemetry: Instrumentation or sampling
- Performance measurement is always non-deterministic (Georges, 2007)
  - Time function is inexact
  - Thread scheduling changes
  - Non-determinism inside of the VMs: Optimization and JIT, garbage collection, class loading, ...
- Research focuses at
  - ... accuracy: Do two measurements yield the same results?
  - ... hot method detection: Where is the most time spent and how to optimize these methods?
    (Mytkowicz, 2010); (Burchell et al, 2024)
- Often, we want to detect regressions
  - Why did my unit / integration / load test got slower?
  - Why did the performance of my production system change?

# Outline

- Observability Techniques: Instrumentation and Sampling
- Change Detection Benchmark
- Benchmarking Result
  - Change Detection Efficiency
  - Tracing Overhead
- Summary and Outlook

# Outline

- Observability Techniques: Instrumentation and Sampling
- Change Detection Benchmark
- Benchmarking Result
  - Change Detection Efficiency
  - Tracing Overhead
- Summary and Outlook

# Instrumentation

- Adding code in order to detect program events

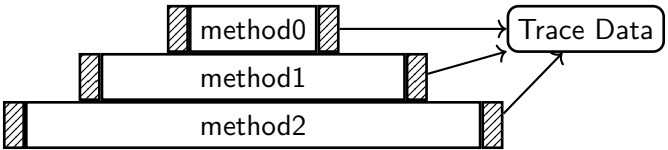
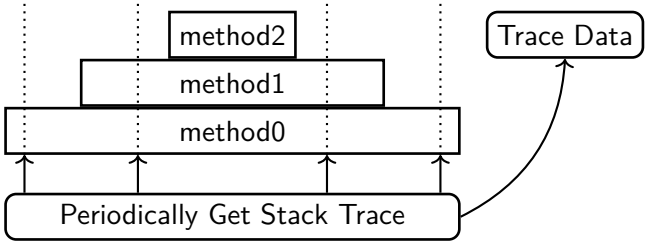



Abbildung: Instrumentation Process (hatched areas are executions of probe code)

- Techniques (Waller, 2015)
  - Manual instrumentation
  - Automated adaption of byte code
    - Aspect-oriented: AspectJ
    - Byte code instrumentation: ASM, ByteBuddy, Javassist
  - Middleware interception

# Sampling

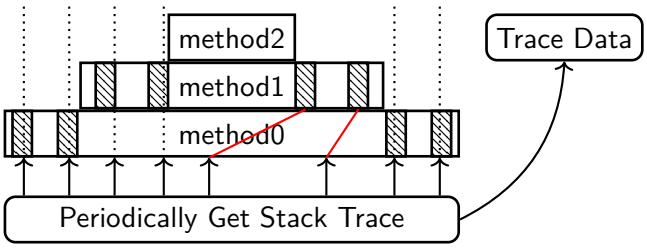



```
void method0   2 ms
void method1   2 ms
void method2   0 ms
```

Abbildung: Sampling Process

- Current state is obtained within defined time intervals
- Might get stack trace, value of a metric, system counter, ...
- Getting stack trace might be triggered by hardware or software timer

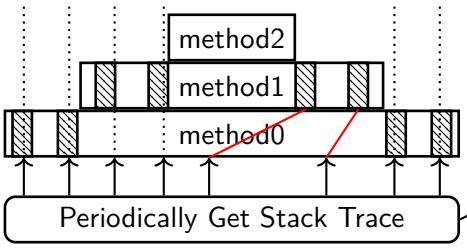
# Safepoint Bias




```
void method0   2 ms
void method1   2 ms
void method2   0 ms
```

- Safepoints (or yield points) occur when no memory operations are performed (Mytkowicz, 2010)
  - Safepoints are costly and the JVM tries to avoid them
  - Safepoints are avoided in very short-running methods
    $\Rightarrow$ Sampling tends to ignore small methods
- Solutions (often only partially documented)
  - perf + perf−map−agent (kernel-Level, requires sudo)
  - JavaMissionControl: Usage of JVM-internal tracing
  - async-profiler: Use HotSpot-specific AsyncGetCallTrace

# Safepoint Bias



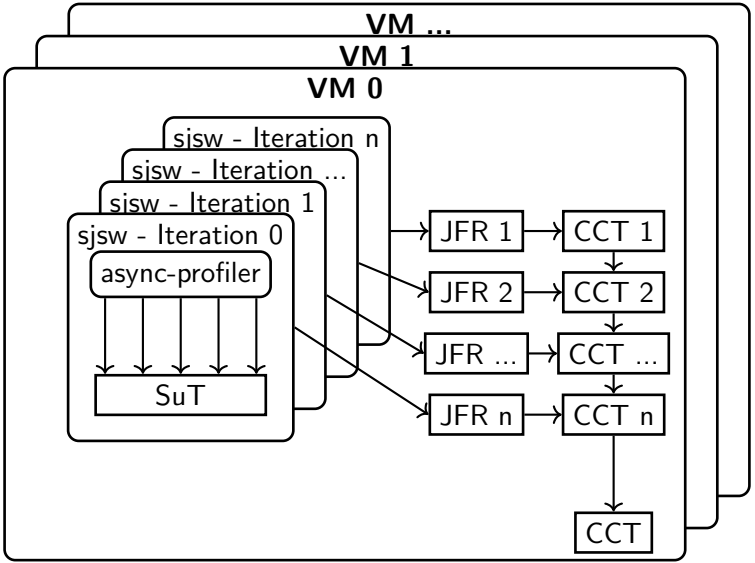

```
void method0   2 ms
void method1   2 ms
void method2   0 ms
```

- Safepoints (or yield points) occur when no memory operations are performed (Mytkowicz, 2010)
  - Safepoints are costly and the JVM tries to avoid them
  - Safepoints are avoided in very short-running methods
    ⇒ Sampling tends to ignore small methods
- Solutions (often only partially documented)
  - perf + perf−map−agent (kernel-Level, requires sudo)
  - JavaMissionControl: Usage of JVM-internal tracing
  - async-profiler: Use HotSpot-specific AsyncGetCallTrace

# SJSW

- SJSW: Simple Java Sampling Wrapper (`https://github.com/terahidro2003/sjsw/`)
- Library that abstracts the details of sampling
- Resulting tree types (Ammons et al., 1997)
  - Dynamic Call Tree (DCT): Preserves all calls
  - Call Graph: Only caller-callee relations ("gprof" problem: metric at call C cannot be attributed to C's callers)
  - **Context Call Tree / Calling Context Tree** (CCT): Reduction of the DCT by considering two vertives equivalent if they represent the same procedure and the parent is equal

# SJSW – Sampling Process



# Outline

- Observability Techniques: Instrumentation and Sampling
- Change Detection Benchmark
- Benchmarking Result
  - Change Detection Efficiency
  - Tracing Overhead
- Summary and Outlook

# Results Analysis Process

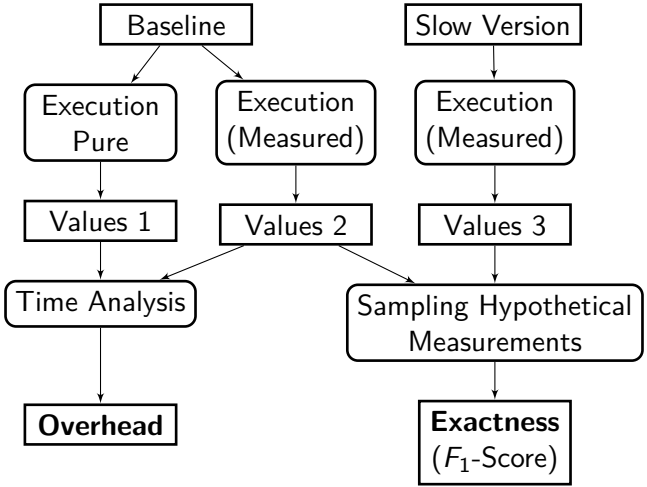



Abbildung: Results Analysis Process

# Project Structure

- Artificial projects are generated for different tree depth (2, 4, 6, 8)
- Sampling is automated using SJSW
- Instrumentation is done using the Kieker observability framework (Yang et al.: "The Kieker Observability Framework Version 2.")

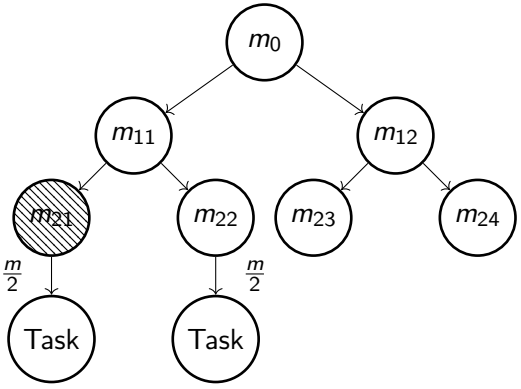



Abbildung: Call Tree Graph ($d = 2$)

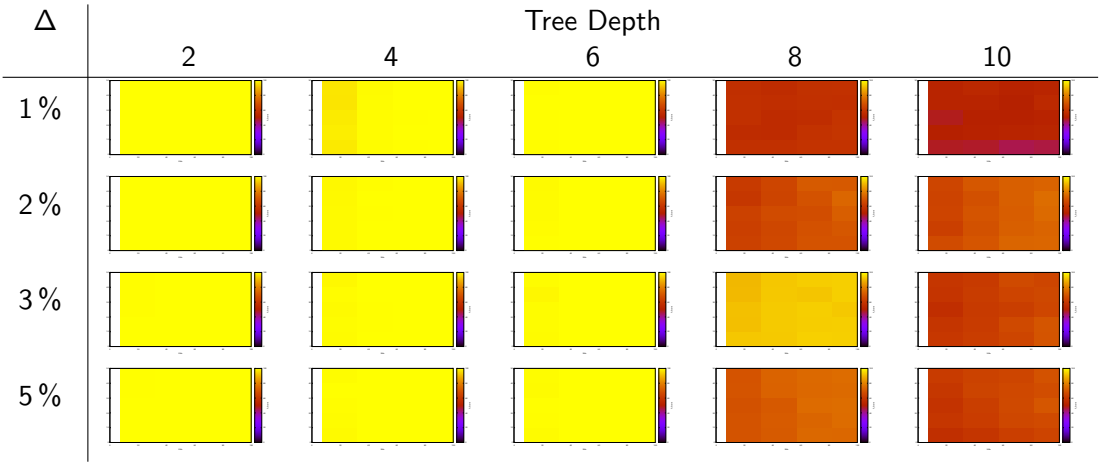
# Outline

- Observability Techniques: Instrumentation and Sampling
- Change Detection Benchmark
- Benchmarking Result
  - Change Detection Efficiency
  - Tracing Overhead
- Summary and Outlook

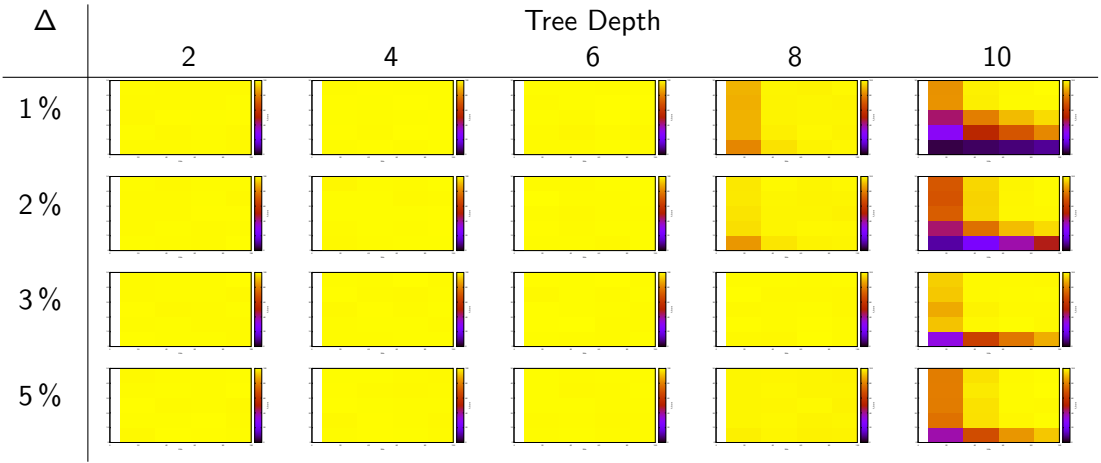
# Outline

- Observability Techniques: Instrumentation and Sampling
- Change Detection Benchmark
- Benchmarking Result
  - Change Detection Efficiency
  - Tracing Overhead
- Summary and Outlook

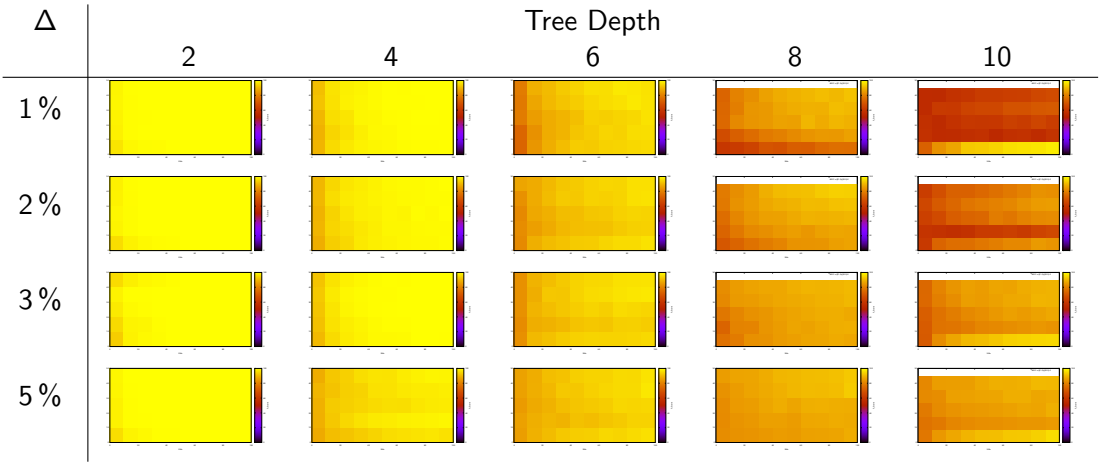
# Instrumentation – T-Test



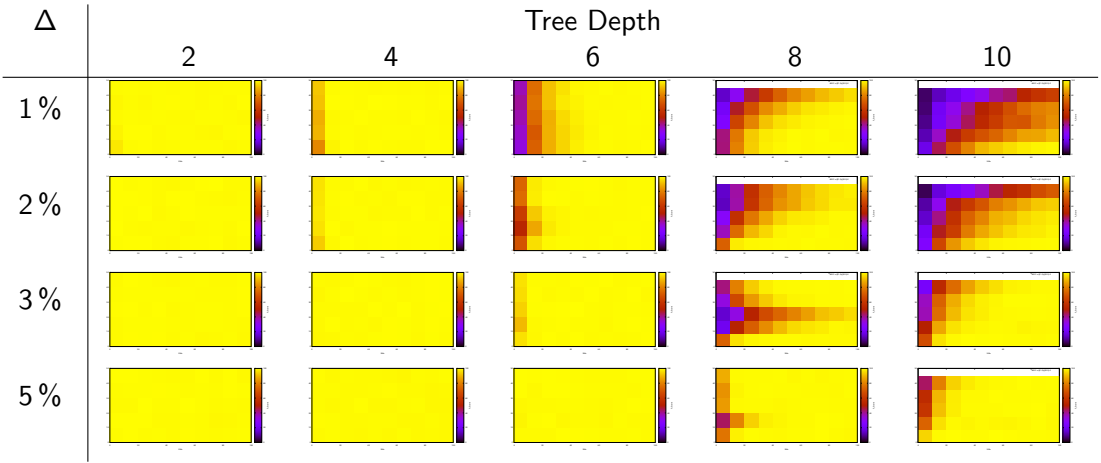
# Instrumentation – Mann-Whitney-Test



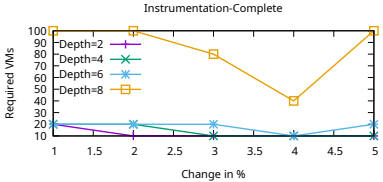
# Sampling (1ms) – T-Test

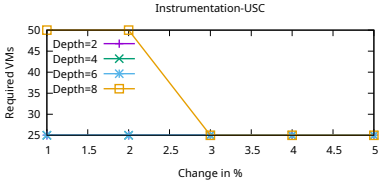


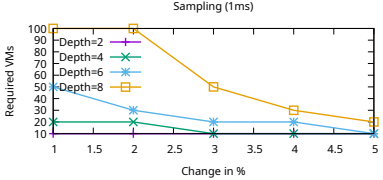
# Sampling (1ms) – Mann-Whitney-Test



# $F_1$-Score Comparison



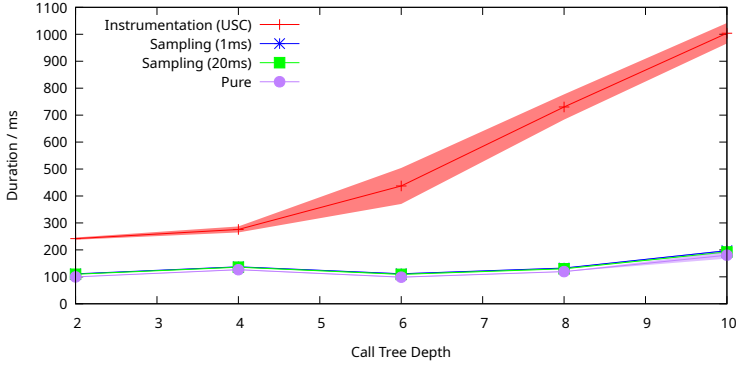
# Outline

- Observability Techniques: Instrumentation and Sampling
- Change Detection Benchmark
- Benchmarking Result
  - Change Detection Efficiency
  - Tracing Overhead
- Summary and Outlook

# Overhead Examination

- MooBench: Benchmark for monitoring overhead (Waller et al.: "Including Performance Benchmarks into Continuous Integration to Enable DevOps", 2015)
  `https://github.com/kieker-monitoring/moobench`


- Overhead examination: "Replicate" MooBench results
- $\mu_6 = \frac{437\mu s - 99\mu s}{(3 \cdot 6+1) \cdot 100000} = 0.178\mu s$
- $\mu_8 = \frac{730\mu s - 119\mu s}{(3 \cdot 8+1) \cdot 100000} = 0.244\mu s$
- $\mu_{10} = \frac{1004\mu s - 180\mu s}{(3 \cdot 10+1) \cdot 100000} = 0.266\mu s$



# Outline

- Observability Techniques: Instrumentation and Sampling
- Change Detection Benchmark
- Benchmarking Result
  - Change Detection Efficiency
  - Tracing Overhead
- Summary and Outlook

# Summary

- Instrumentation and sampling are techniques that make it possible to detect performance differences
  - Instrumentation inserts code
  - Sampling uses existing extension points
- Benchmarking of change detection efficiency and overhead
  - Artificial project generation
  - Measurement with sampling, instrumentation, and without anything
- Benchmarking results
  - Sampling has low overhead and good exactness
  - Complete instrumentation is unrealistic
  - Instrumentation should only be used if single requests need to be traced

# Outlook

- Integration of SJSW into Kieker
  - Would be more a async-profiler – Kieker integration
  - Requires significant technical work
- Measurement for microservice projects
  - "Real" projects most likely have different characteristics than MooBench / precision-experiments-rca
- Understanding of the JVM compilation behavior
  - Experiments have sometimes surprising results
  - Explaining these results (automatically) might be interesting and helpful

# Minimal Overhead Instrumentation [WOSP-C '23]

- Default Kieker (and OTel) instrumentation extracts more information than necessary
- Optimized instrumentation can reduce the overhead from $4.77\mu s$ to $0.39\mu s$ per method invocation

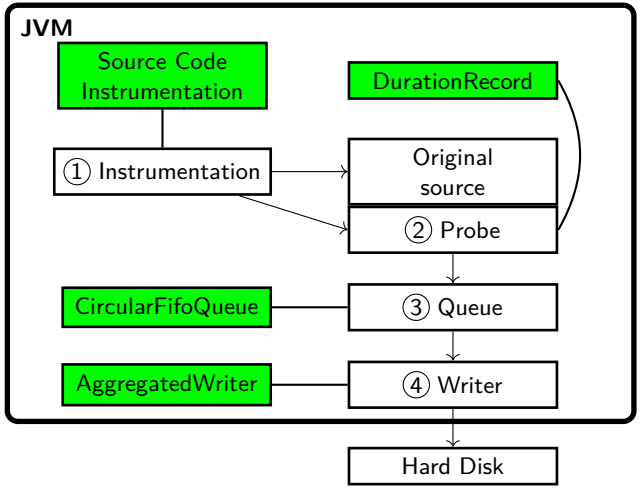
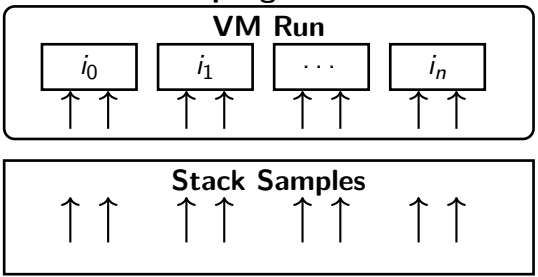



Abbildung: Monitoring Optimizations (for **minimal** overhead ;-) )

# SJSW – Sampling Modes

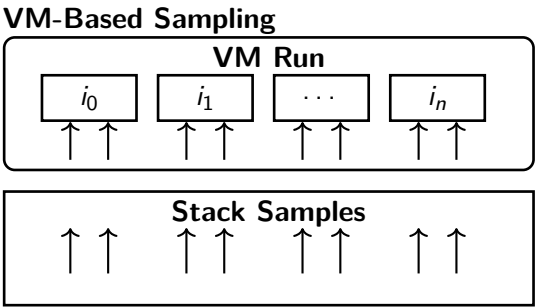
- VM-based sampling: Whole VM is sampled, by adding −javaagent to the VM
- Iteration-based sampling: Sampling is activated and deactivated programmatically

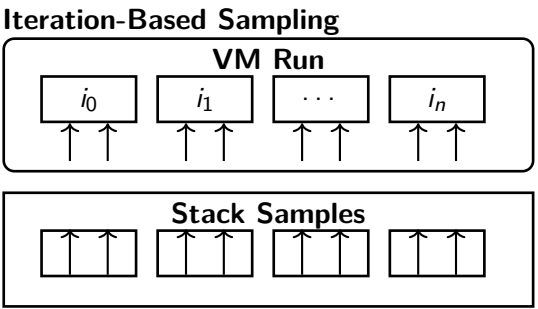
**VM-Based Sampling**



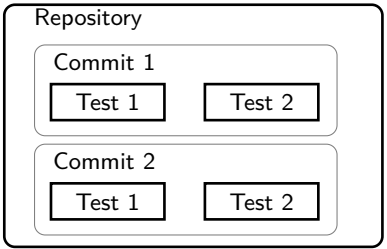
# SJSW – Sampling Modes

- VM-based sampling: Whole VM is sampled, by adding −javaagent to the VM
- Iteration-based sampling: Sampling is activated and deactivated programmatically

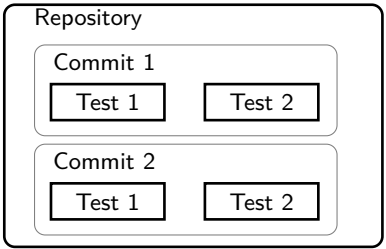


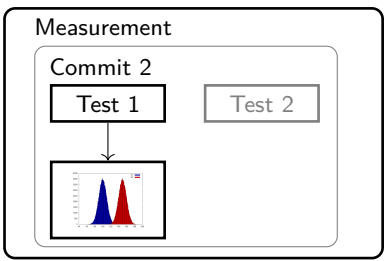



# Process of Peass

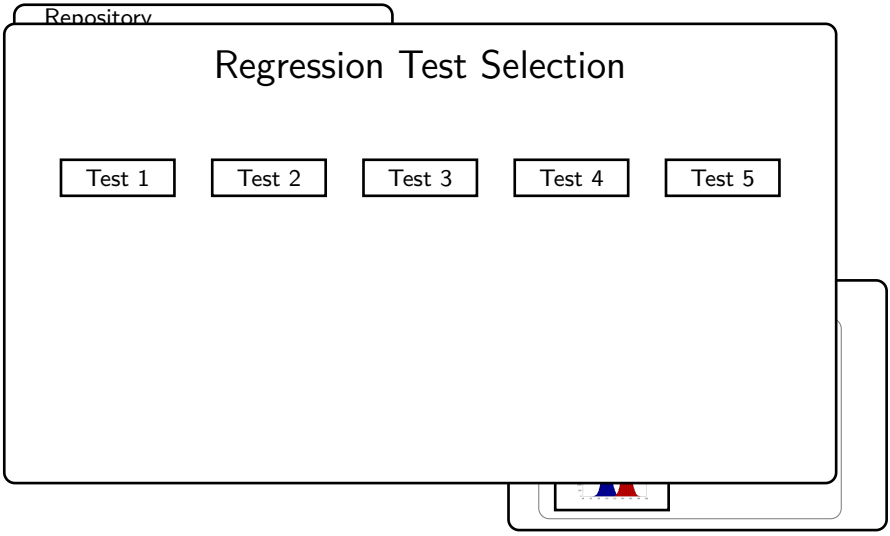


# Process of Peass

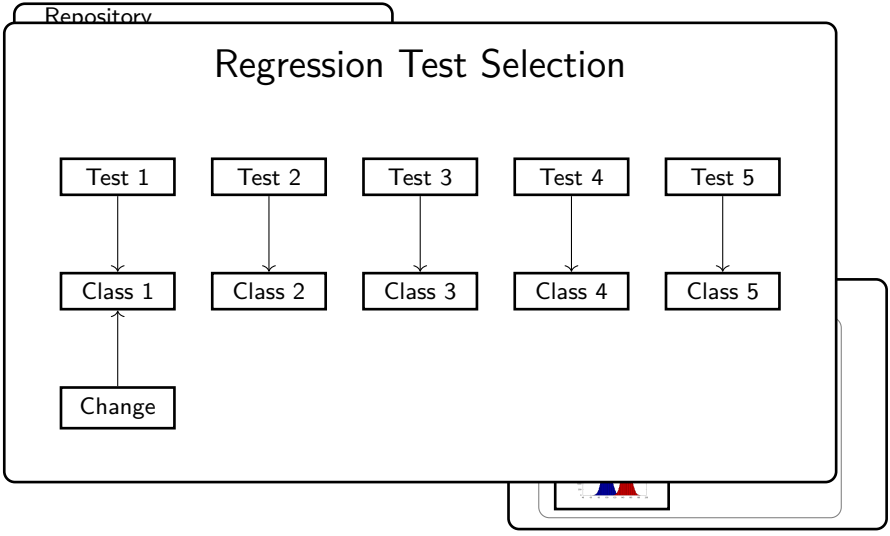


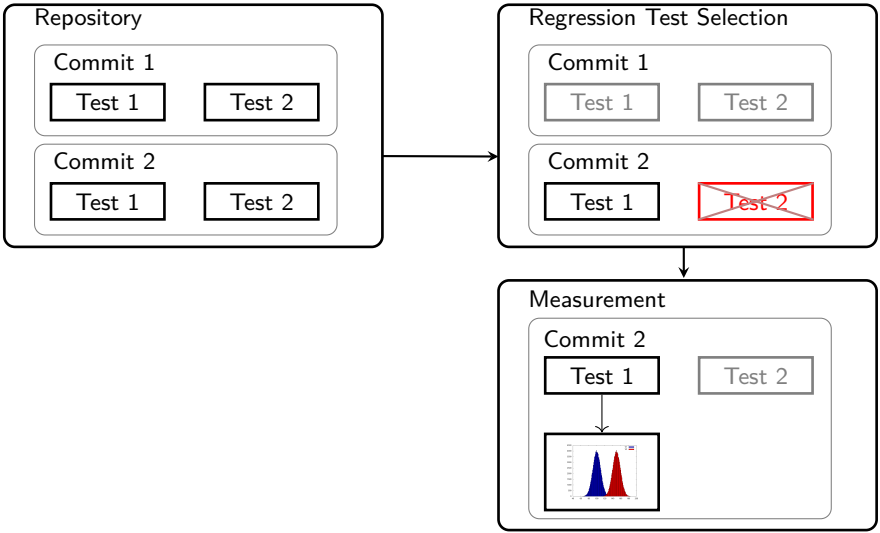
# Process of Peass



# Process of Peass



# Process of Peass



# Process of Peass