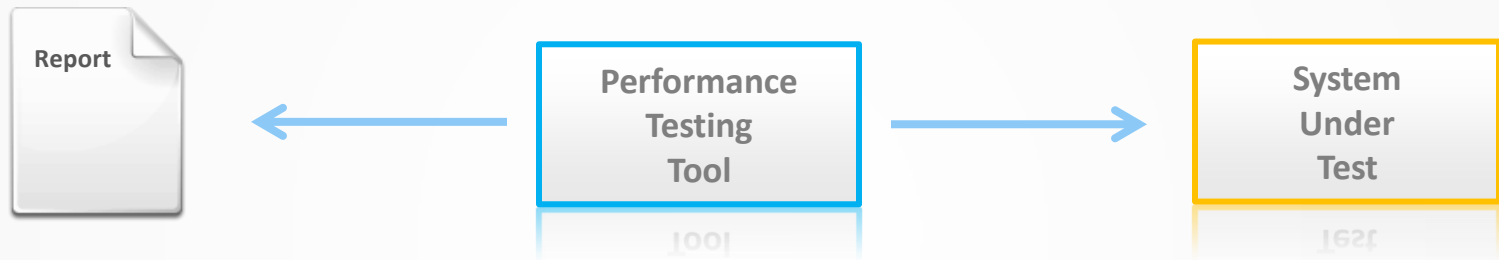


JAGGER

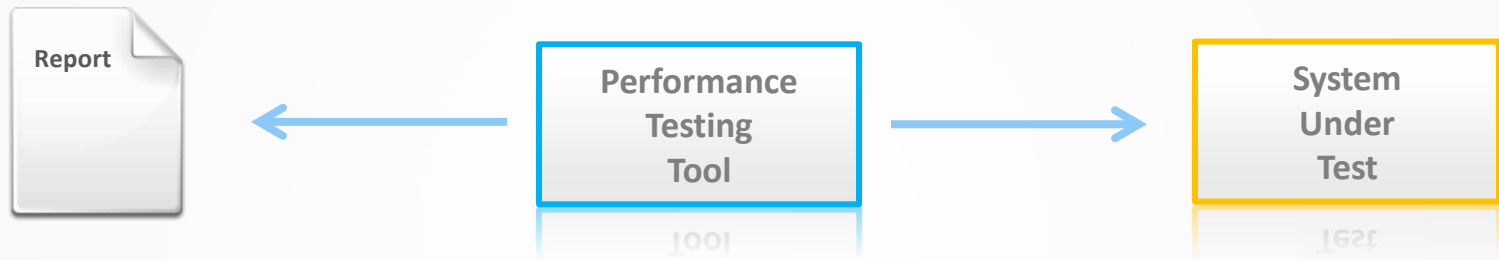
R1

INDUSTRIAL-STRENGTH  
PERFORMANCE  
TESTING

# PREFACE



**What's wrong with this schema?**



**What's wrong with this schema?**

**Nothing. But it is not easy to make it work correctly.**

# Typical Story :: 1

**Delivery Manager:** New version goes live soon. We need to check that performance meets SLA.

2 days passed

**Performance Tester:** Performance degraded in new release in comparison with the previous. Please investigate.

**Technical Lead:** Let's review changes...

5 hours passed

**Technical Lead:** Joe, one of your commits looks suspiciously.

**Developer:** I need to check. I'll run it locally under profiler.

3 days passed

**Developer:** I found non-optimal code and fixed it. Profiler shows 150% increase in performance. Please retest.

2 days passed

**Performance Tester:** Performance increased on 10%, but it's still worse than in the previous release.

**Delivery Manager:** We can't wait any more. Our CPU utilization in production is 50% only, let's go with what we have.

Performance testing should be continuous. If performance degrades, alerts should be raised.

If something wrong with performance, it should be easy to identify cause and reproduce the problem.

# Typical Story :: 2

**Operations:** We deployed new release into production and it passed all smoke tests. Responses became incorrect when we switched load balancer to it.

**QA Lead:** We tested all this functionality.

**Performance Test Lead:** We tested that system has acceptable throughput under workload, there were no failed transactions, there were no memory leaks.

2 days passed

**Technical Lead:** We investigated this and it turned out that one singleton unit is not thread safe.

It is not enough to test that system works under workload. Performance test should check that it works correctly.

# Typical Story :: 3

**Performance Tester:** Performance degraded in the last release candidate. What happened?

**Technical Lead:** We merged branches A and B to the trunk for this release. And a couple of fixes from branch C. What exactly impacted performance?

**Performance Tester:** I never heard about branch B. We tested A, but that tests are incomparable with the current tests for trunk. And can't find last report for branch C. What a mess...

**Performance Testing Tool is not enough for large projects. Such projects need a comprehensive performance testing process and Performance Testing Server that tracks all results.**

# Typical Story :: 4

**VP of Engineering:** Production system failed this morning. What happened?

**Technical Lead:** It looks like somebody restarted two nodes simultaneously. After this synchronization failed and all the cluster went down.

**Operations:** We noticed that two nodes consumed too much memory and restarted them.

**VP of Engineering:** Did you test this scenario on pre-prod environment?

**QA Lead:** We tested similar scenario two months ago, but this is a complex manual test, we can't do it continuously.

**Robustness and fail over should be tested not only in production. Simulation of maintenance operations and failures should be a part of non-functional testing.**



# JAGGER OVERVIEW

# Performance Testing, Revised

- **Continuous:** Performance Testing Server automatically tests all builds and rises alerts in case of performance degradation
- **Traceable:** There is a master database that stores all results of testing and provides ability to browse and compare them
- **Transparent:** Modules can be tested in isolation. Results of performance testing include monitoring and profiling information, so it is possible to understand *structure of performance* directly from reports.
- **Validated:** Performance testing provides some level of guaranties that system behavior under workload is functionally correct
- **Pessimistic:** Performance testing scenarios include simulation of potential maintenance operations and failures

**There is no framework that fits this model**

# Impact on Jagger Architecture

- **Continuous** and **Traceable** properties imply high level of automation and configurability, high volume of stored data, automatic decision making capabilities.
- **Transparent** property assumes automatic monitoring and profiling capabilities for distributed environments, higher volumes of data, firewall-friendly communication protocols.
- **Validated** property assumes ability to check results of interaction with System Under Test and extra workload for Jagger itself.
- **Pessimistic** property assumes tools for network and system management that can be orchestrated with performance testing process.

# Jagger Applicability

Besides the mentioned concepts, Jagger is designed to support all standard types of performance testing:

## Stress Testing

Distributed workload generators can create very high workload

## Spike Testing

Workload can be specified as a function of time, hence spikes can be modeled

## Endurance Testing

Distributed storage can handle data collected during long-running tests

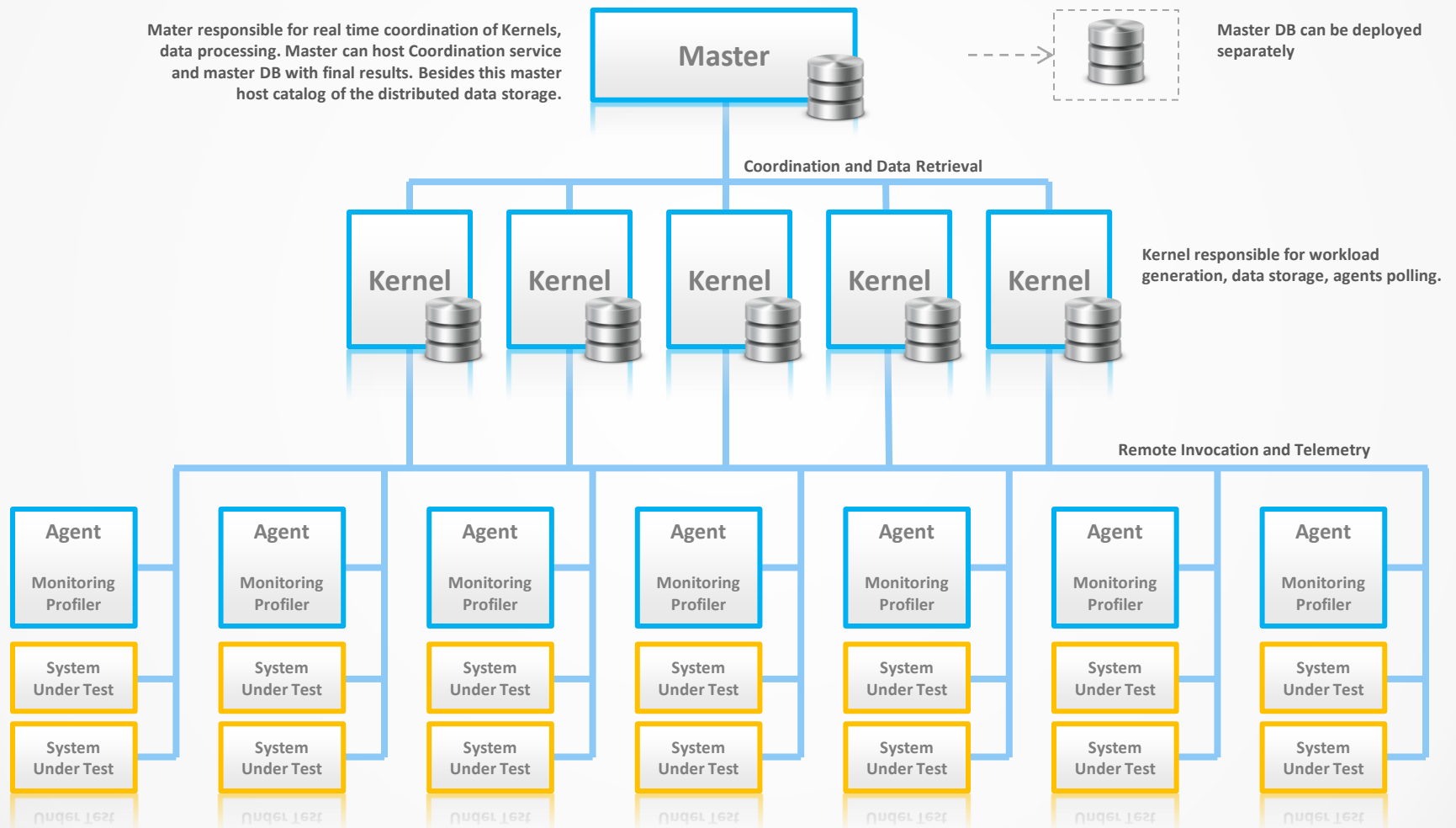
## Isolation Testing

A library of primitives for remote services invocation facilitates isolated components testing

## Load Testing

A number of workload generation strategies and comprehensive reporting and statistics

# Bird's Eye View on Jagger



Jagger deployment typically consists of three types of units: Master, Kernel, and Agent. Minimal deployment is so-called local mode that runs Master and Kernel in one JVM. At the other extreme, Master is deployed on a dedicated machine, user feeds it with test configuration and receive reports from it. Master coordinates a farm of Kernels that generate workload and collect telemetry from Agents. Agents are optionally deployed on boxes with System Under Test and each Agent is able to monitor multiple SuTs on one box.

In the next section we will discuss Jagger from the user perspective, and after that we come back to its architecture and technologies underneath it.

# FEATURES

# Automation :: Continuous Testing

All Jagger flows are completely automatic and can be launched from within Hudson, Maven and about any other CI or build system

All final results are persisted to RDBMS, hence can be browsed using variety of tools and history can be traced

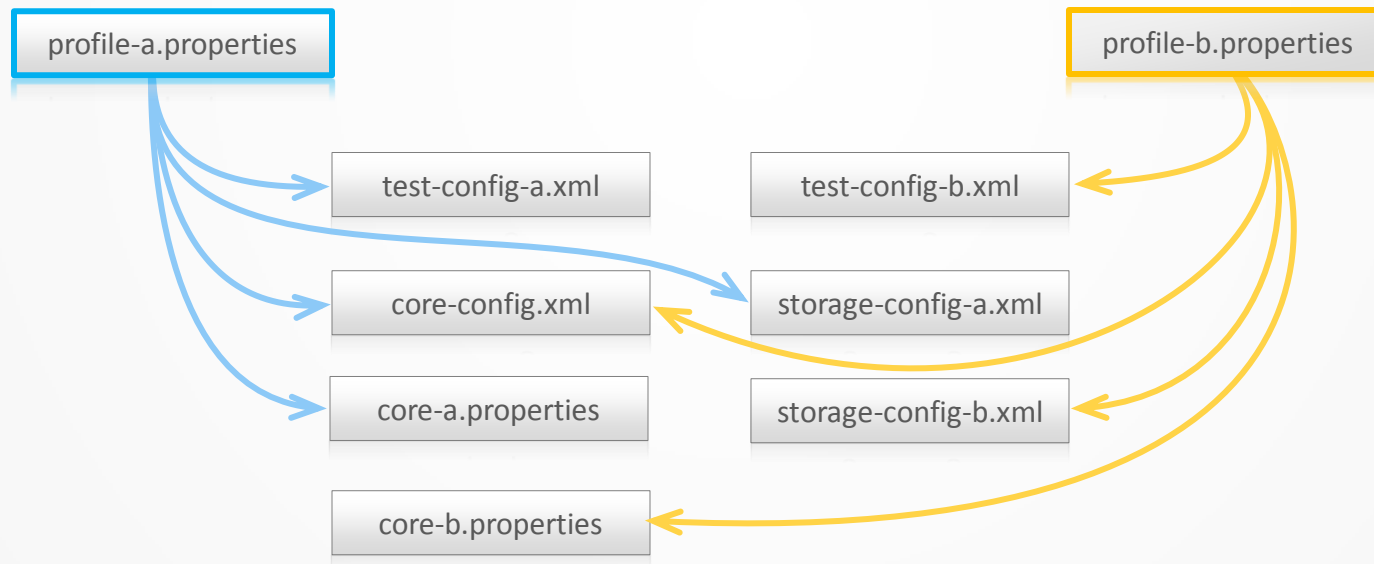
The screenshot displays the JaggerDB on VM interface. On the left, a sidebar shows the database structure under 'Local Database Servers' > 'JaggerDB on VM' > 'Databases' > 'jaggerdb' > 'Tables' > 'E1ScenarioData'. The main window shows a table with 15 rows of test results. The table has columns: id, avgLatency, failuresCount, kernels, samples, samplesPerThread, sessionId, stdDevLatency, successRate, taskid, threadCount, and throughput. The table is titled 'Table Data' and 'Primary Key - jaggerdb..E1ScenarioData'. The status bar at the bottom indicates 'Pattern:' and 'Total Rows: 404'.

	id	avgLatency	failuresCount	kernels	samples	samplesPerThread	sessionId	stdDevLatency	successRate	taskid	threadCount	throughput
1	1	0.12	0	2	10	5.3		0.03	1	task-1	2	6.52
2	2	2.82	0	2	10	5.3		0.76	1	task-2	2	0.35
3	3	0.11	0	2	30	5.3		0	1	task-3	6	22.78
4	4	2.04	0	2	30	5.3		0.84	1	task-4	6	1.4
5	5	0.11	0	2	40	5.3		0	1	task-5	8	31.85
6	6	1.63	0	2	40	5.3		0.59	1	task-6	8	2.13
7	7	0.11	0	2	50	5.3		0	1	task-7	10	38.58
8	8	1.5	0	2	50	5.3		0.25	1	task-8	10	3.19
9	9	0.11	0	2	80	5.3		0	1	task-9	16	59.93
10	10	1.55	0	2	80	5.3		0.28	1	task-10	16	4.78
11	11	0.12	0	2	10	5.4		0.02	1	task-1	2	5.19
12	12	2.92	0	2	10	5.4		1.47	1	task-2	2	0.34
13	13	0.11	0	2	30	5.4		0	1	task-3	6	22.01
14	14	2.12	0	2	30	5.4		0.86	1	task-4	6	1.27
15	15	0.11	0	2	40	5.4		0	1	task-5	8	32.84

Pattern: Total Rows: 404

# Automation :: Profiles Management

Each test project or environment can use completely independent set of components and Jagger cluster properties



Jagger offers advanced configuration system. This system can be briefly described as follows. One can shred configuration into arbitrary set of XML and properties files, some of these files can be shared between different test configurations or environments, and some can be test- or environment-specific. Each test specification has a root properties file that contains path masks of other pieces. Elements of configuration can be located in different folders or disks. Jagger will automatically discover all these pieces, apply property overriding and substitution rules and assemble final configuration that specifies both test scenarios as well as configuration of Jagger core services.



# Automation :: Decisions Makers

A pluggable system of decision makers allows to compactly present testing statuses and automate human effort for report inspection

Session Status

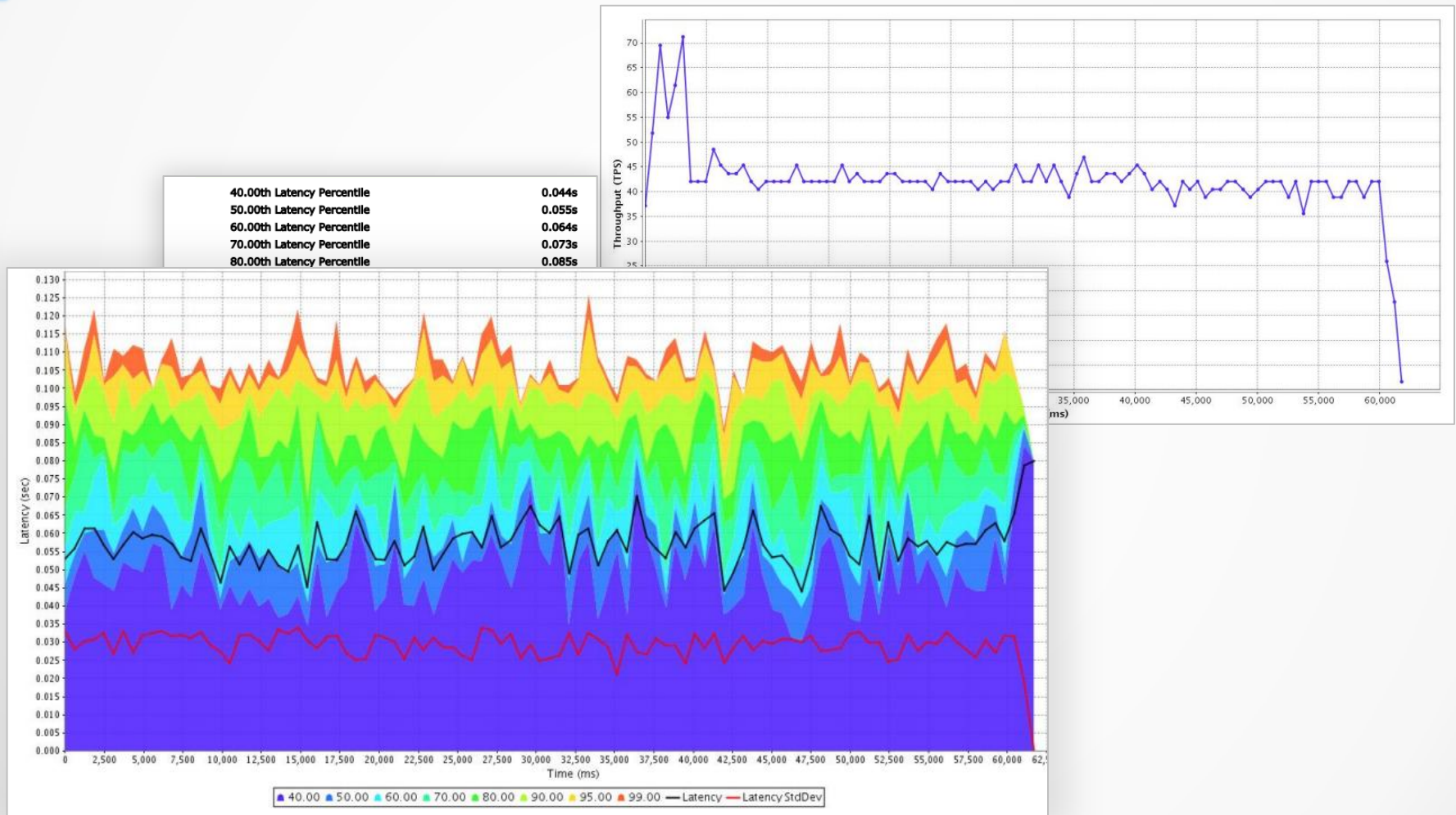
Status is based on success rate. Session status is a status of the worst test.

Test Summary (E1 Engine)

Scenario Name	Clock	Throughput (TPS)	Details	
<div>sleep-service-10ms---terminate-after-60-seconds</div> <div></div>	1 virtual	58.85	Version	1
			Samples	3672
			Termination Strategy	Terminate after 60 seconds
			Duration (sec)	62.000
			Average Latency (sec)	0.020
			StdDev Latency (sec)	0.010
			Number of failures	0
			Success Rate	1.00
<div>sleep-service-50ms---terminate-after-60-seconds</div> <div></div>	1 virtual	17.37	Version	1
			Samples	1061
			Termination Strategy	Terminate after 60 seconds
			Duration (sec)	61.000
			Average Latency (sec)	0.060

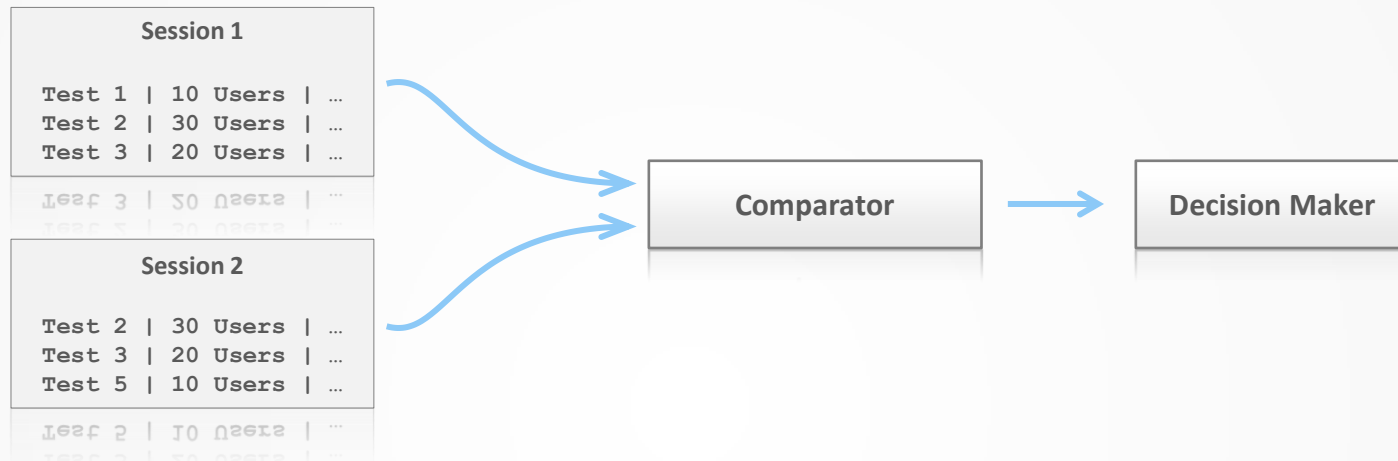
Testing results for each test individually and for entire session are routed to the decision makers. This allows to map entire session to a single status that clearly indicates that results are acceptable or not. Jagger is bundled with the configurable decision makers, but one can easily write a custom decision maker in Java, Groovy, or JRuby.








# Time Series and Statistics



Jagger collects and reports comprehensive statistical information for all tests. This information includes both total scalar values as well as time plots that provide insights into test dynamics.

# Session Comparison

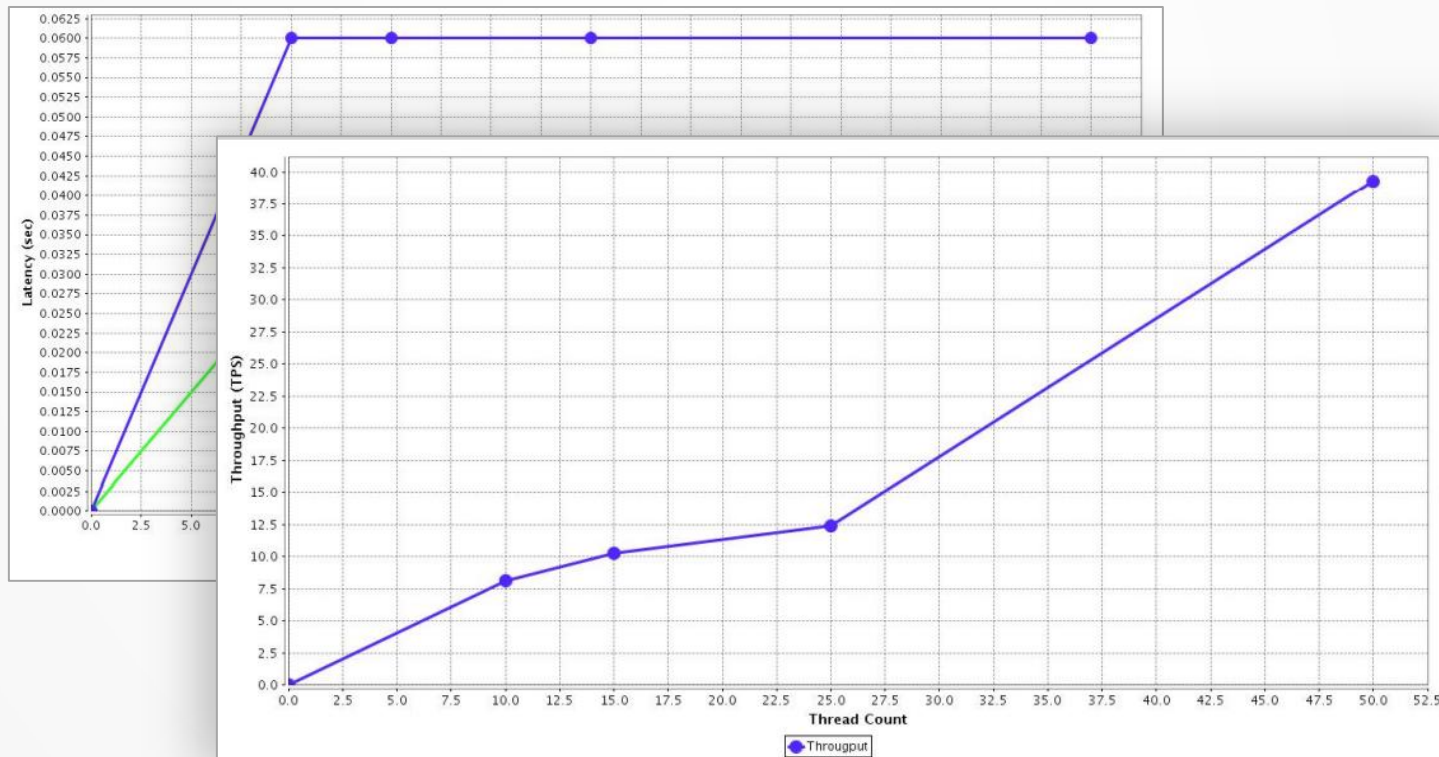


 Current Session : 70 Baseline Session : 66								
Scenario Name	Scenario Version	Thread Count	Throughput Ratio	Total Duration Ratio	Mean Latency Ratio	StdDev Latency Ratio	Success Rate Ratio	Status
d-google-page	1	1	-80.18%	∞%	4475.00%	1200.00%	0.00%	
d-gd-page	1	1	46.15%	-40.00%	-41.50%	-4.29%	0.00%	
d-google-page-time	1	1	-98.63%	30.00%	6866.67%	9700.00%	0.00%	
d-gd-page-time	1	1	-22.86%	-15.38%	28.77%	-40.82%	0.00%	
d-google-page	1	2	-80.00%	3400.00%	6900.00%	7200.00%	0.00%	
d-gd-page	1	2	13.79%	-16.67%	-15.83%	-20.91%	0.00%	

Jagger is able to automatically compare results of performance testing with a baseline. A baseline can be specified as a result of previous testing session (say, previous release build) or as a manually created set of values. Results of session comparison are routed to the decision maker that transform numerical deviation from the baseline to the status (acceptable deviation, warning, etc).

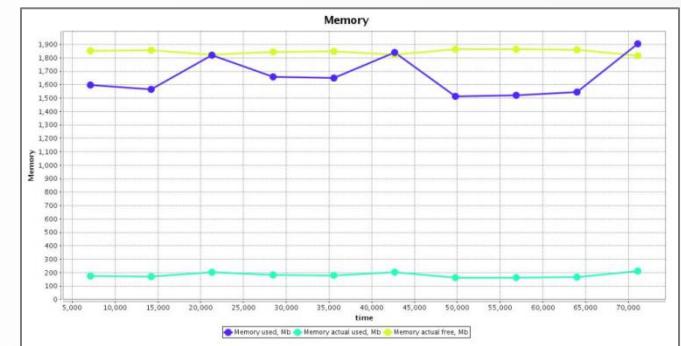
# Scalability Analysis

Reports contain plots that consolidate results of several tests that differs in workload and, consequently, visualize system scalability:



# Monitoring :: 1

Jagger has embedded monitoring system and provides comprehensive telemetry in reports:

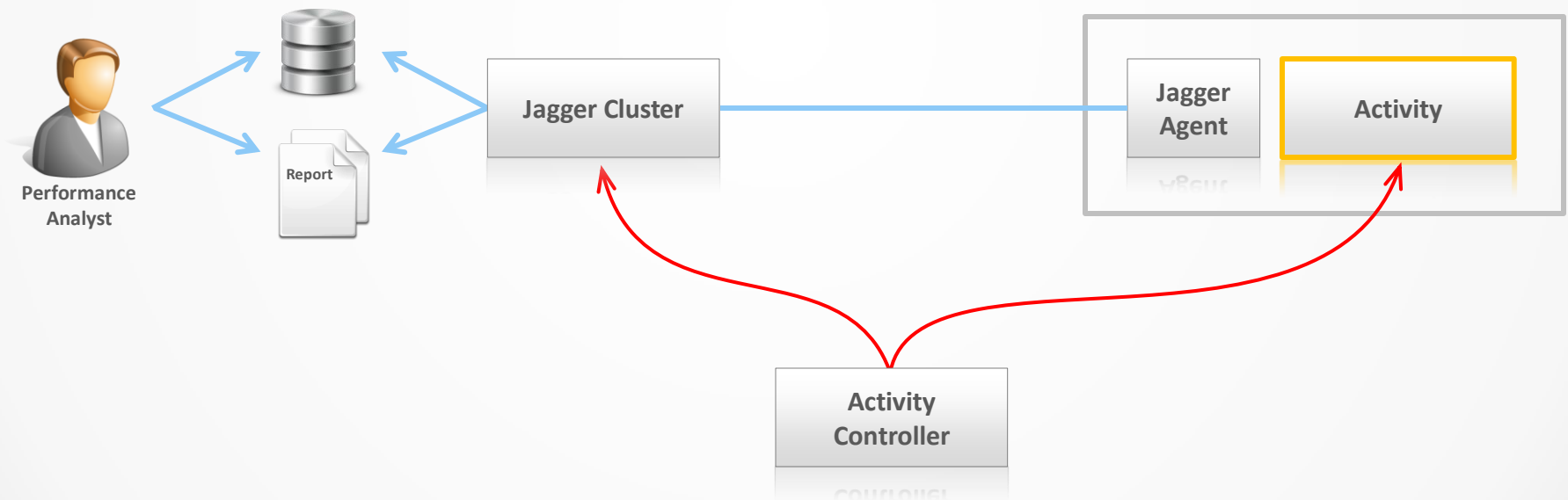


Jagger communicates with monitoring agents via HTTP/Bayeux protocol which is transparent for firewalls and port restrictions

Jagger is bundled with monitoring agents which are based on cross-platform SIGAR monitoring library. SIGAR enables Jagger to collect variety of system metrics, besides this Jagger is able to poll JMX to collect JVM-specific parameters. Jagger is designed to support large number of monitoring agents and collect/store significant amount of monitoring data. Core Jagger communicates with Agents via Bayeux protocol, so there is no necessity even in additional open port on Agent side. For example, both system under test and Agent can use port 80 and that's it.

# Monitoring :: 2

Jagger can be used as a pure monitoring tool, separately from performance testing. One typical use case is monitoring of regular activities:



Jagger monitoring system is not coupled with performance testing – one can use Jagger only for monitoring. Of course, Jagger is not a generic monitoring system like Cacti or Zabbix, but it can be used to track performance or impact of regular activities/processes. For example, if some activity (data backup, heavy analytical query etc.) is executed periodically, activity controller can trigger both activity and Jagger Monitoring. In this case Jagger will collect telemetry for each activity execution, produce reports and persist collected data to its DB. This information can be used to track performance of the activity and its impact on system workload.

# Workload Generation

**Workload generation strategy is pluggable, it can be**

- Static (for example, fixed number of virtual users)

- Dynamic, controlled by feedback from Kernels  
(for example, increase throughput until response time is below threshold)

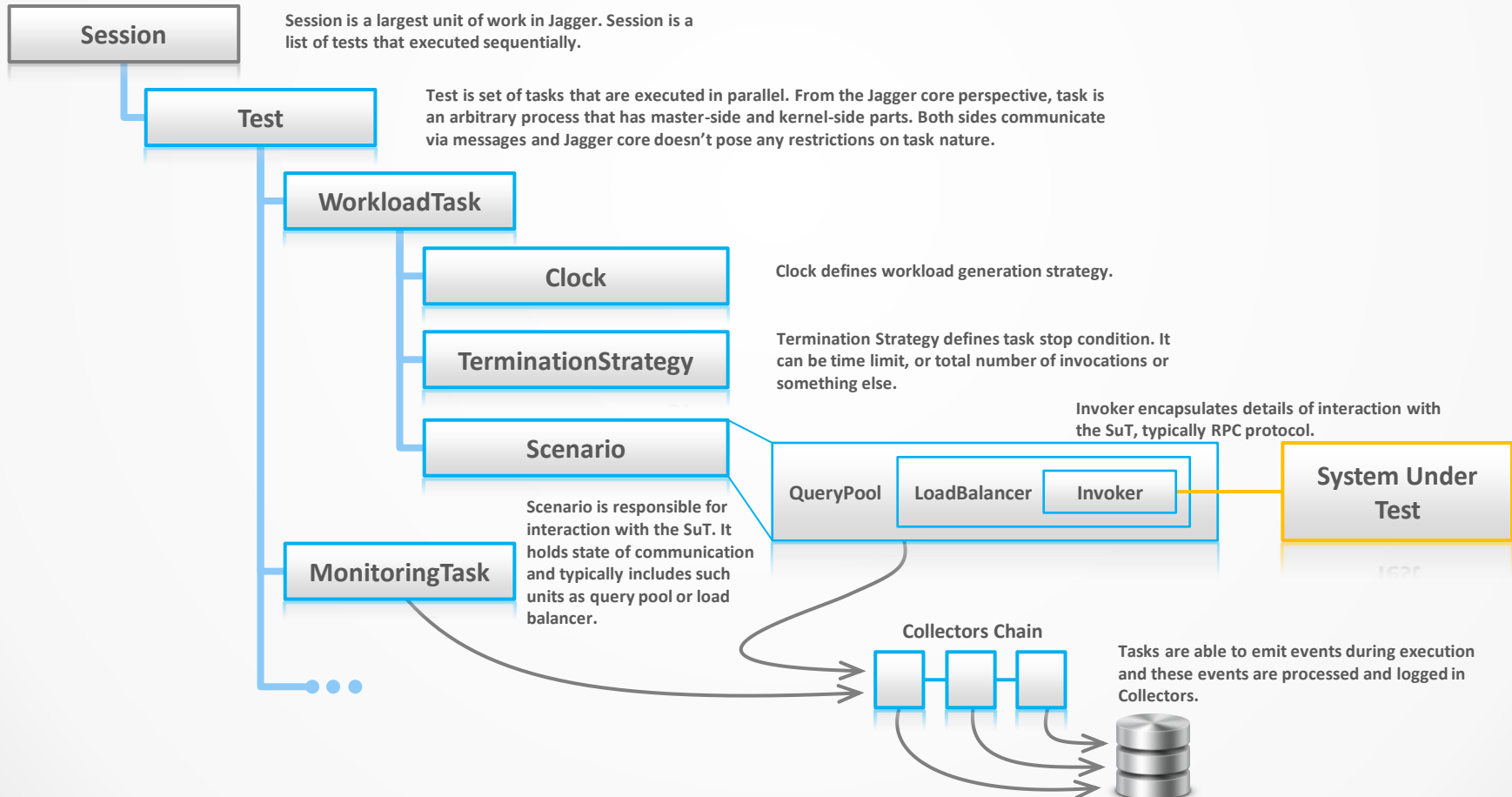
**Two strategies are provided out of the box:**

- Virtual users with configurable behavior

- Total throughput specified as an arbitrary function of time. In particular, this allows to model workload spikes.

# User API :: Extensibility

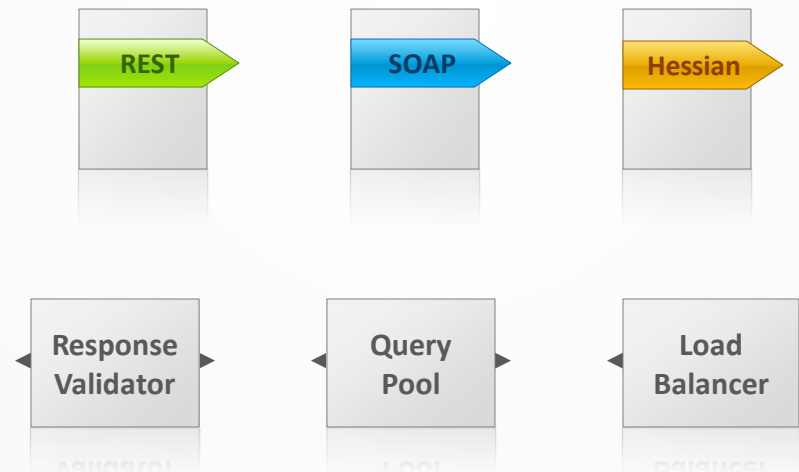
Jagger has an extensible hierarchy of configuration entities. Any block highlighted in **blue** can be overridden in user-specific way:





# User API :: Protocols and Primitives

A number of invocation protocols and primitives are supported out of the box:



Jagger is bundled with a library of ready-to-use primitives for building workload test scenarios. It includes support of several RPC protocols and typical units of logic like software load balancer or test scenario with a query pool.

# User API :: Validation

Jagger provide ability to validate responses of System Under Test in two ways:

- An invocation listener can be added to perform custom checks

- If query pool is known before test, Jagger can automatically collect responses for each query in single-user fashion before test and then check that system returns the same responses under workload.

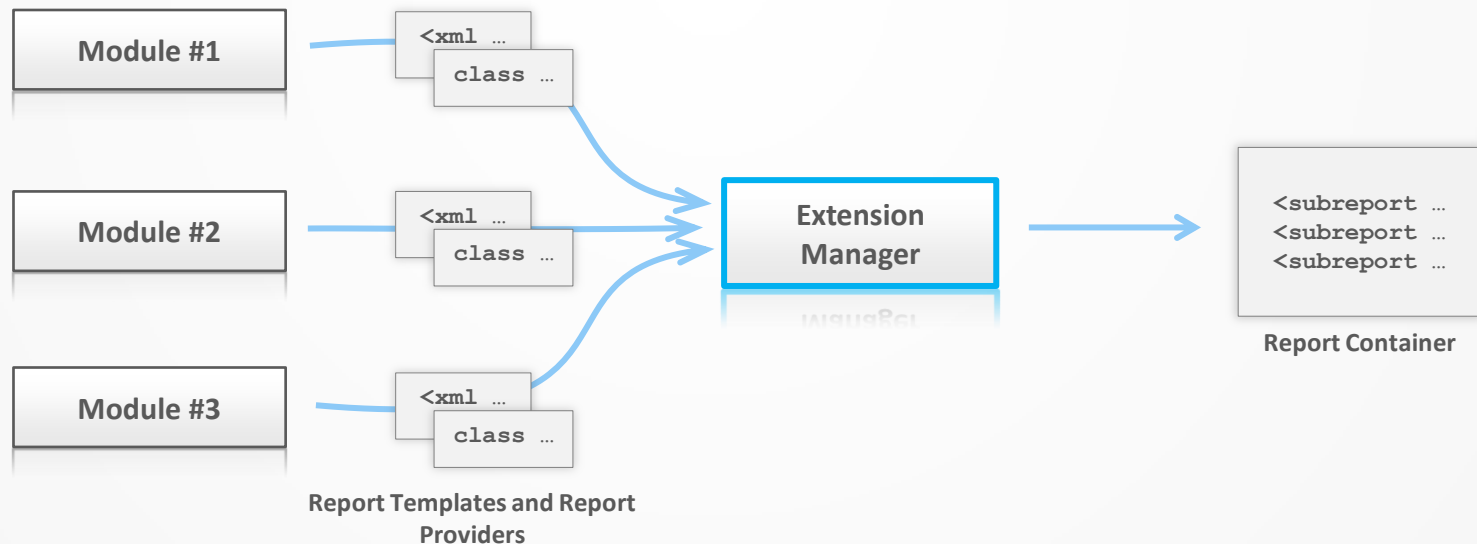
# Extensibility and Dynamic Languages

- Any module can be overridden in XML configuration files and implemented in Java, Groovy or JRuby
- Zero-deployment for test scenarios in Groovy: scenario can be modified without Jagger cluster restart

Jagger heavily relies on Spring Framework for components wiring. All XML descriptors are externalized and editable, so one can override any module. New implementation can be either written in Java or Groovy/JRuby sources can be inserted directly in XML descriptors. Jagger not only allows to write test scenarios in Groovy, but also able to run such scenarios in distributed cluster without cluster restart or redeployment.

# Flexible Reporting

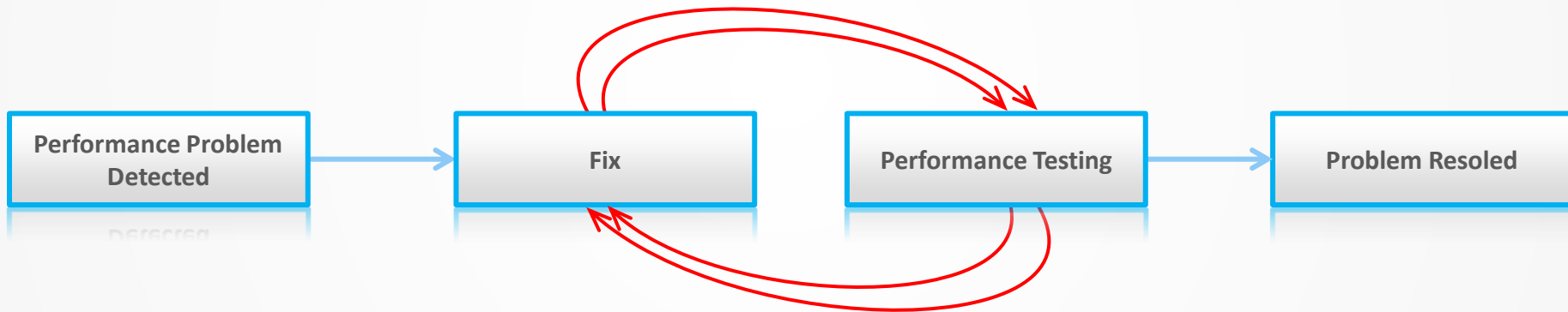
- Support PDF and HTML output formats
- Reporting system is based on JasperReports
- Pluggable design and fully customizable report structure
- Reports can be generated separately from testing



Reporting system is based on JasperReports – well known and mature solution for reporting. All report templates (in XML format ) are externalized and editable, so operator is able to configure report layout, change look-and-feel, include/exclude/modify any sections. Jagger provides an Extension Management mechanism that allows to register new report templates and Report Providers that supply templates by data. Report Providers can be written in Java, Groovy or JRuby. As soon as extension is registered, it can be included into hierarchy of report containers as a subreport. Typically report is generated after each test session, but Jagger provides ability to generate report based on DB data separately from testing.

# Embedded Profiler :: 1

## Performance issues investigation – a typical flow:



It is not enough to detect insufficient performance or performance degradation. Without insights into system under test it is difficult for developers to reproduce the problem and fix it reliably. Developers often try to profile a problematic unit on their local environments but such results are often distorted, especially for complex server-side applications. In practice this means many fix-test round trips and time losses. Jagger has an embedded profiler that provides insights into application and hints for performance problem investigation.

# Embedded Profiler :: 2

Jagger provides sampling profiler for JVM that works via JMX and doesn't require JVM agent

Profiling results are included into reports

Profiler contains intelligence to detect hot spots

Observation count : 27603  
Methods registered : 482

Hot Spots:

Method	Self Time Ratio	In Stack Ratio
org.apache.zookeeper.ClientCnxn\$SendThread#run	16.99%	16.99%
org.mortbay.io.nio.SelectorManager\$SelectSet#doSelect	11.32%	11.32%
org.mortbay.io.nio.SelectorManager#doSelect	0.00%	11.32%
org.mortbay.jetty.nio.SelectChannelConnector#accept	0.00%	11.32%
org.mortbay.jetty.AbstractConnector\$Acceptor#run	0.00%	11.32%
org.mortbay.thread.QueuedThreadPool\$PoolThread#run	0.00%	11.32%
org.apache.hadoop.ipc.Server\$Responder#run	11.32%	11.32%

Profiling  
results

Performance Problem  
Detected

Fix

Performance Testing

Problem Resoled

# Distributed Platform

- **Workload generation is distributed. Master continuously polls Kernels statuses and adjust workload if necessary.**
- **Data storage is distributed. This provides both scalability and write performance due to data locality.**
- **Monitoring Agents are supervised by Kernels, so high number of systems can be monitored without bottleneck in a single receiver of monitoring information.**

# Features Summary

## Reporting

- ✓ All data are stored in RDBMS
- ✓ Reports can be generated separately from testing
- ✓ Customizable report templates
- ✓ PDF and HTML reports

## Analysis and Statistics

- ✓ Test status decision making
- ✓ Time plots with statistics
- ✓ Scalability analysis
- ✓ Test sessions comparison
- ✓ Ability to collect and store high data volumes

## Monitoring

- ✓ Embedded monitoring system
- ✓ OS-level telemetry
- ✓ JVM-level telemetry
- ✓ Cross-platform and firewall-transparent
- ✓ Monitoring without performance testing

## Automation

- ✓ Fully automated test execution
- ✓ Profiles management

## Workload and User API

- ✓ Distributed workload generation
- ✓ Customizable virtual users
- ✓ Workload as a function of time
- ✓ Open configuration and high extensibility
- ✓ Java, Groovy, and JRuby support
- ✓ Dynamic deployment of Groovy test scenarios
- ✓ REST, SOAP, Hessian out of the box
- ✓ Response validation and other primitives

## Profiling

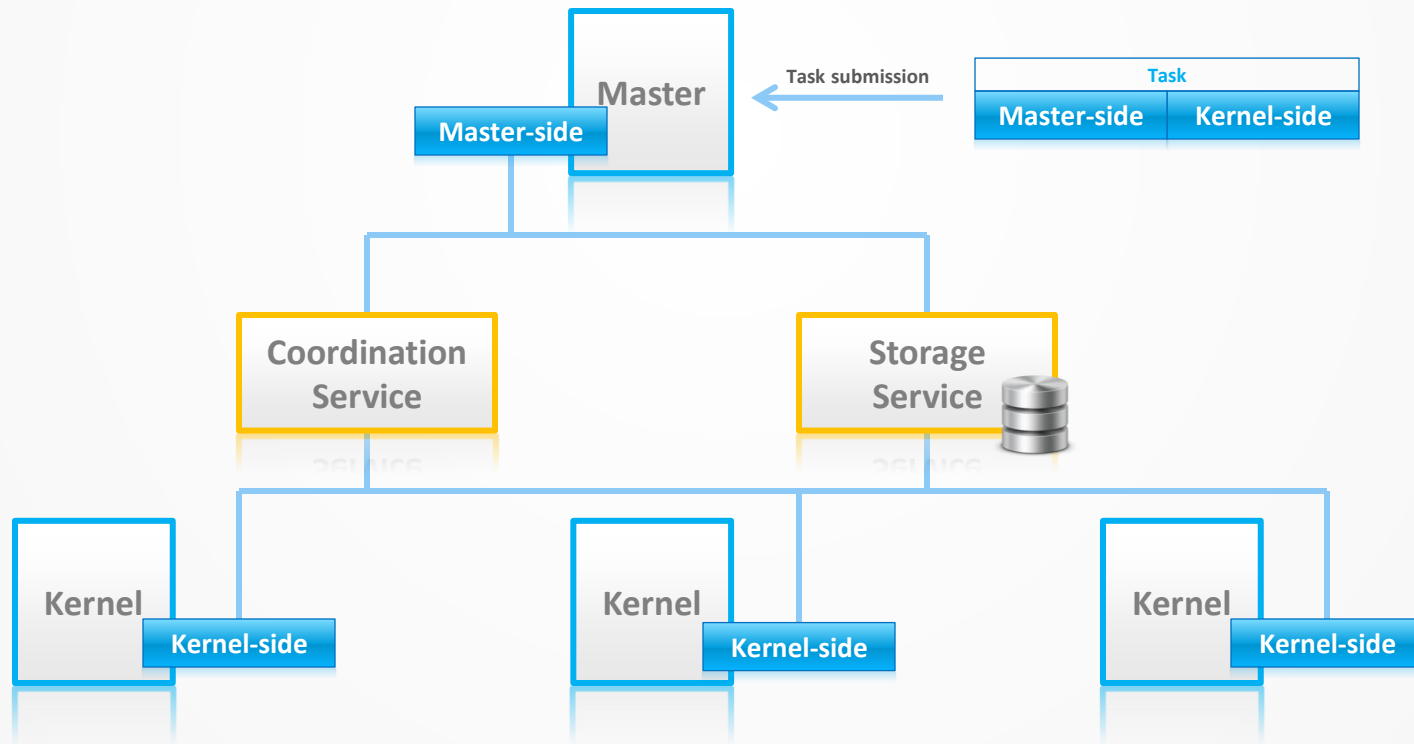
- ✓ Sampling profiler with configurable overhead
- ✓ Automatic hot spots detection



DESIGN

# Core Structure

In its core, Jagger is a quite generic platform for distributed task execution with near-real-time coordination and distributed storage.



Jagger can be considered as platform for distributed task execution, storage and processing of data that are produced by these tasks. Task is almost arbitrary unit of work that should have Master and Kernel side. When tasks is executed Jagger start both sides and these sides can communicate via messages using Coordination service which is part of the platform. It is assumed that tasks communicate one with other using “Data Blackboard” pattern, i.e. one task write its outcome to the Storage service which is also a part of the platform and another task can consume output of preceding tasks as its input.

# Storages

Jagger has three different storage services with different properties and use cases. All storages are shared, Master and Kernels have coherent vision of storages.

## File Storage

**Stores logs, traces, binary data.**

Huge data volume, high performance (10k-100k records/sec/kernel, bandwidth 10MB/sec/kernel)

## Key-Value Storage

**Basically, it is string-to-string map. Stores intermediate results during tests execution.**

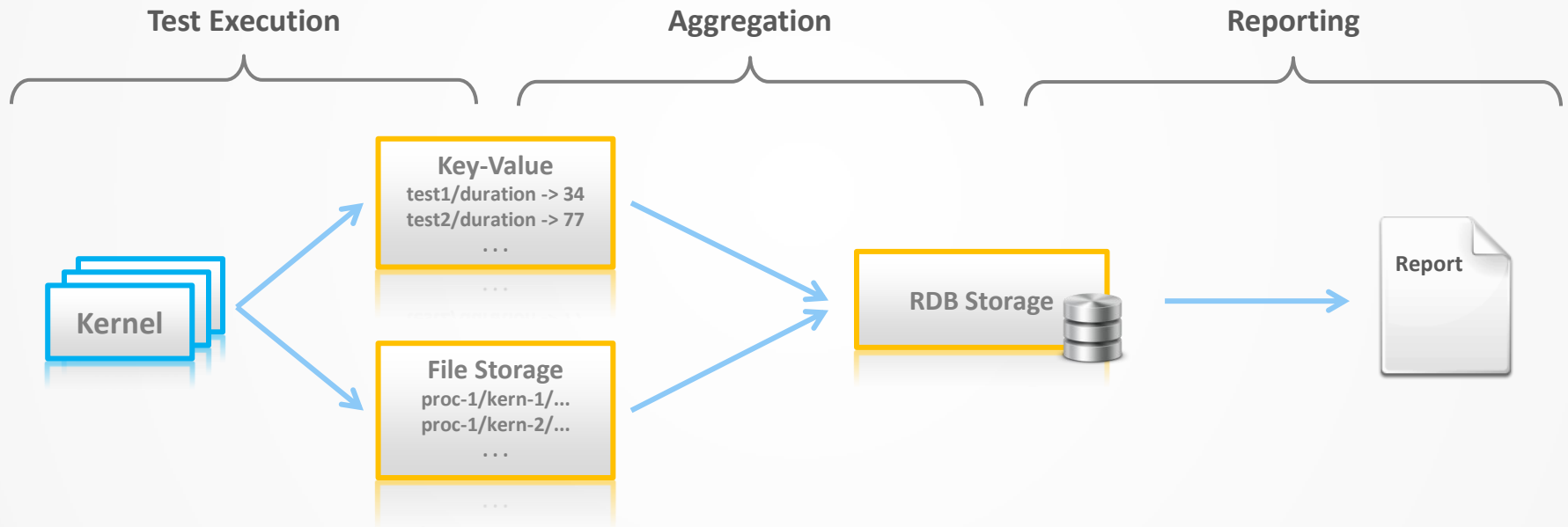
Small data volume, performance is not significant.

## RDB Storage

**Just RDBMS. Stores final results in normalized form and can be used as an integration point with other systems. Operator can customize DB schema.**

Average data volume, performance is not significant.

# Data Flow



From performance testing perspective, data flow consist of three stages. At the first stage, Kernels generate workload and collect monitoring information. All results go to Key-Value and File storages in raw form, separately for each Kernel. At the second stage, all collected data are aggregated (e.g. results from different Kernels are merged and averaged) and outcomes are written to relational database. Finally, at the last stage, aggregated results are rendered and report is produced.

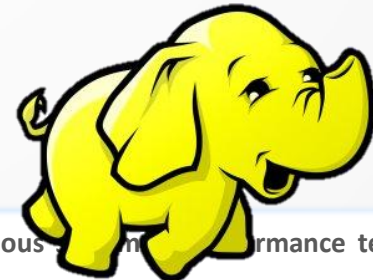
# Data Scalability :: 1

- Total volume of collected data can be high (latency traces, monitoring)
- Total write rate can be high (record for each transaction)
- It should be possible to perform custom analysis of the recorded raw data

Jagger is initially designed to be a distributed Performance Testing Server. This assumes continuous stream of performance testing jobs, high volume of data that are generated with high rate during tests.

# Data Scalability :: 1

- Total volume of collected data can be high (latency traces, monitoring)
- Total write rate can be high (record for each transaction)
- It should be possible to perform custom analysis of the recorded raw data



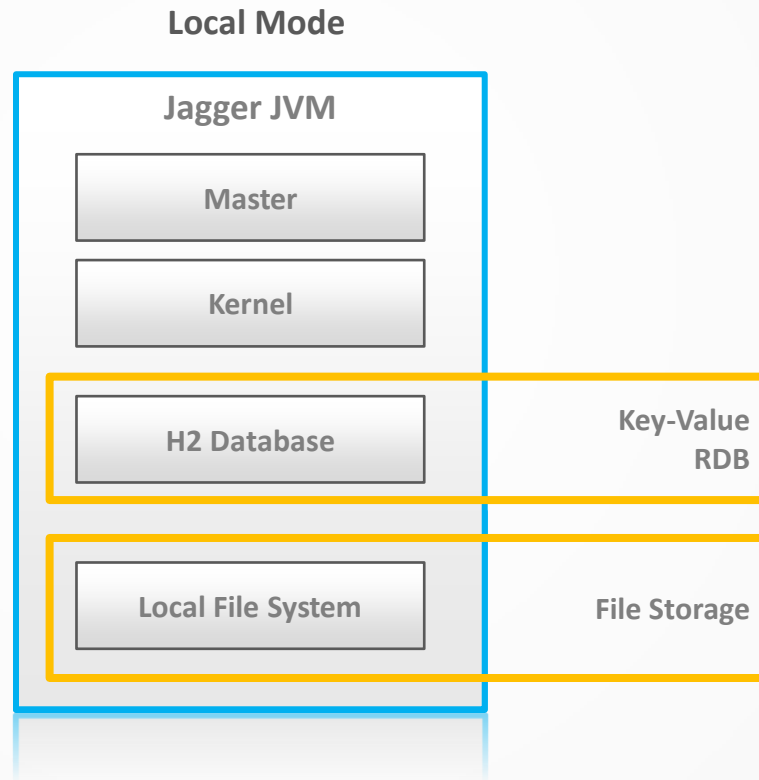
Jagger is initially designed to be a distributed Performance Testing Server. This assumes continuous performance testing jobs, high volume of data that are generated with high rate during tests.

# Data Scalability :: 2

- Jagger uses HDFS as a distributed data storage
- Each Kernel hosts a Data Node
- External Hadoop deployment can be used as a storage and raw data can be processed by means of MapReduce

Jagger uses embedded HDFS to store data that potentially can have high volume (latency traces, monitoring telemetry). Each Jagger Kernel hosts HDFS Data Node that provides very high write rate (data go directly to the local file system) and practically unlimited scalability (tens or even hundreds of Kernels). Jagger can use external Hadoop deployment (collocated with Jagger nodes or not) as a data storage. In this case one can use any technologies from Hadoop stack to analyze and process collected data.

# Deployment Schemas :: 1

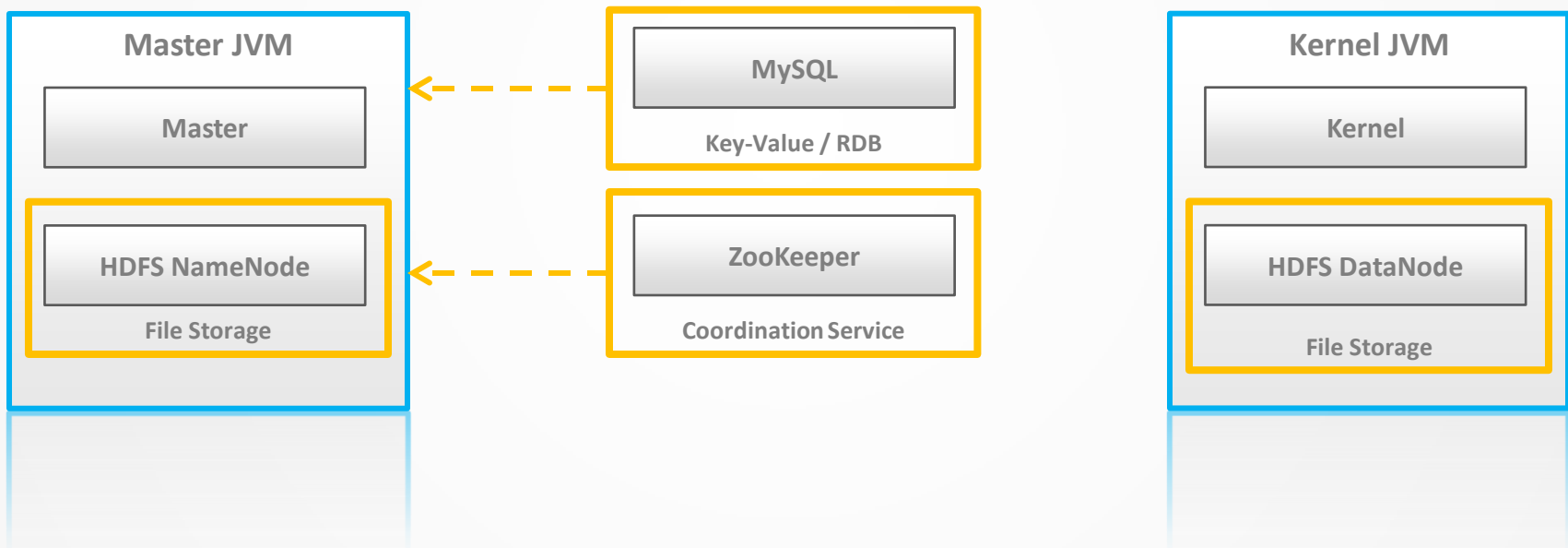


Jagger supports a number of deployment modes which differs in workload and data scale. The simplest schema is Local Mode. Master and Kernel components are started in a single JVM together with RDB and Key-Value storage services which are built on top of embedded H2 Database. File storage is backed simply by local file system.



# Deployment Schemas :: 2

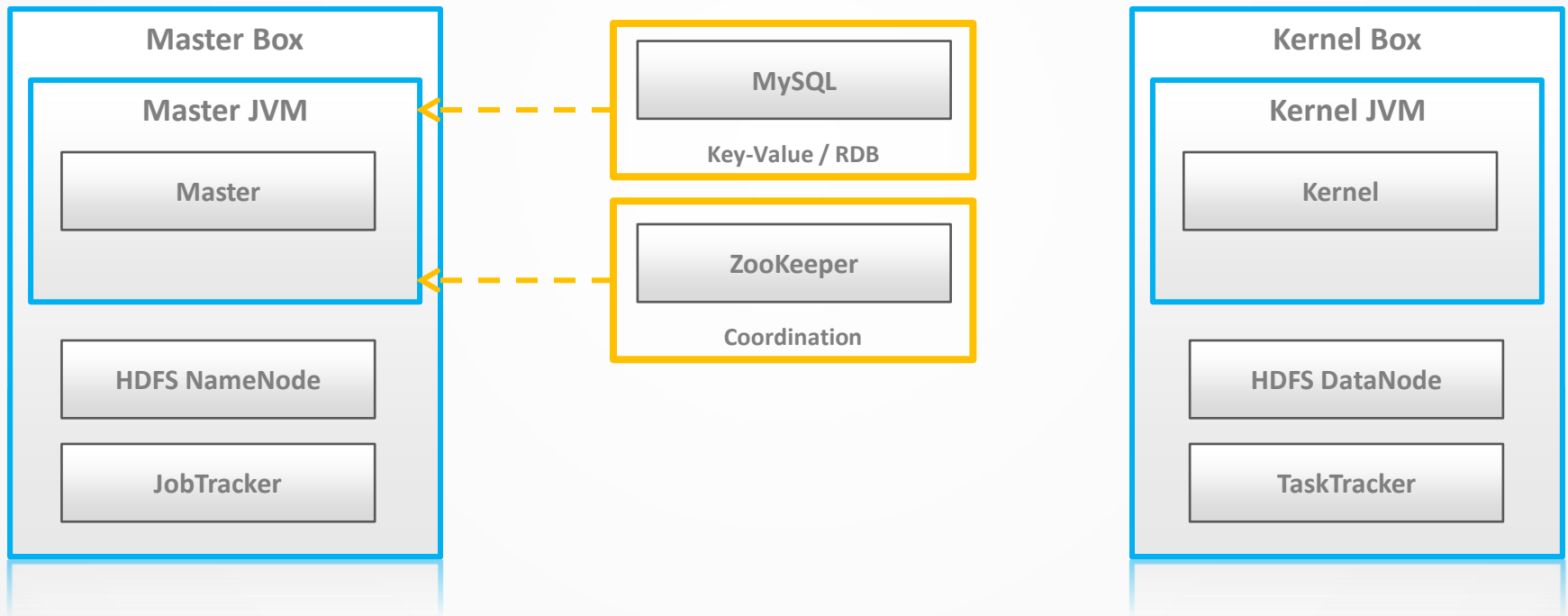
## Distributed Mode



In distributed mode Master can be deployed with a number of Kernels. File storage in this case is distributed and backed by HDFS. Coordination service which is based on ZooKeeper can be deployed both outside Jagger as well as inside Master. Relational database also can be external (MySQL is typically used) or embedded.

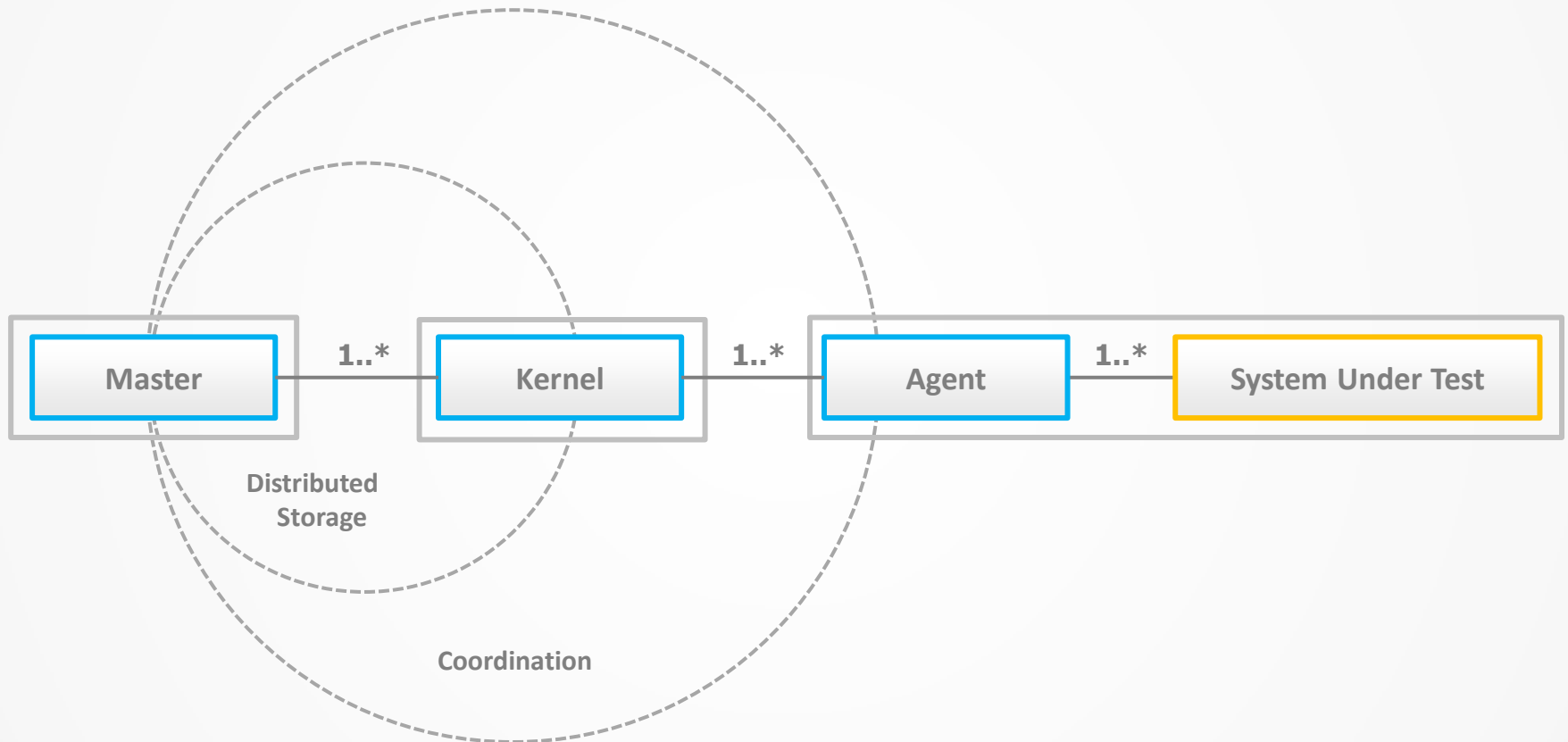
# Deployment Schemas :: 3

## External Hadoop



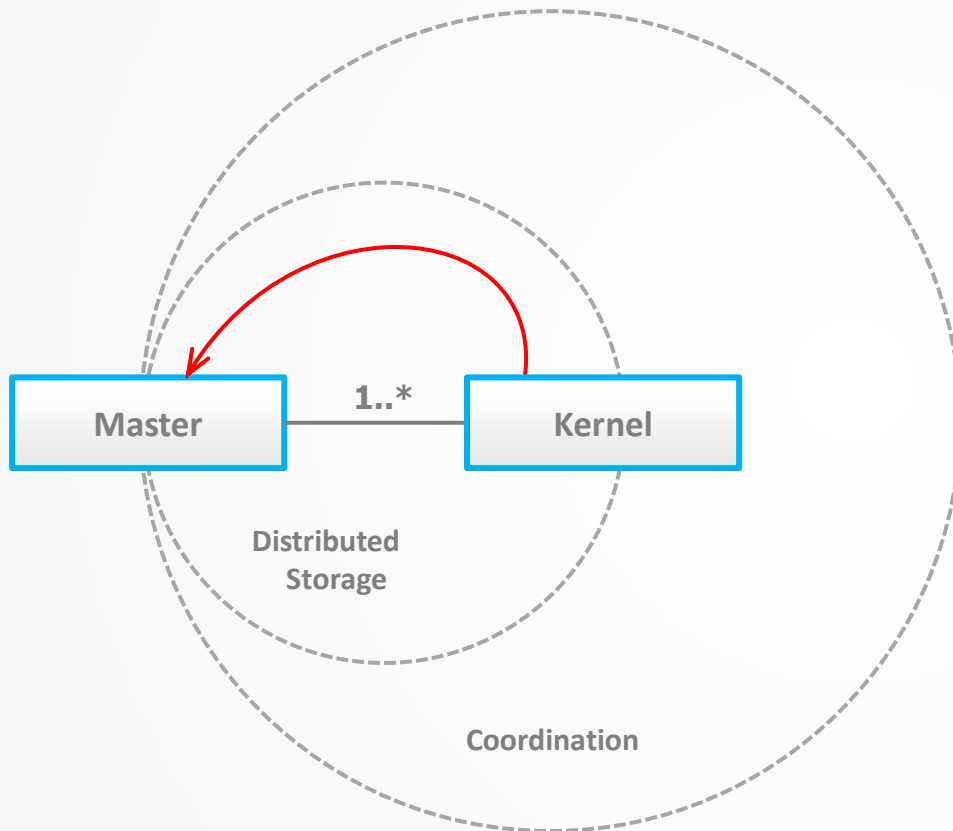
Finally, Jagger can be pointed to external HDFS/Hadoop system, preferably collocated with the Jagger deployment. In this case collected data can be processed outside of Jagger.

# Jagger Cluster :: 1



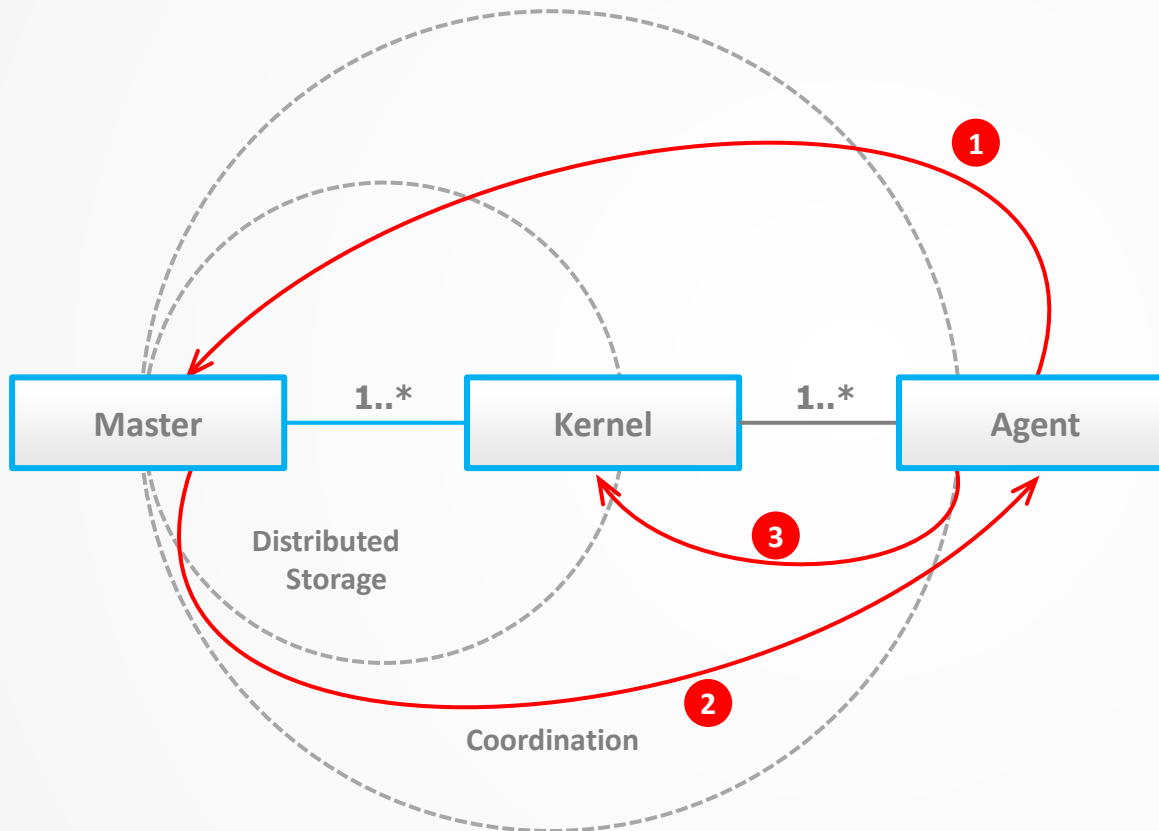
Jagger cluster structure is as follows. There is one Master that can be connected to one or more Kernels. Each Kernel can supervise one or more monitoring Agents and each Agent, in its turn, can collect information about one or more systems under test. Master and all Kernels use common distributed storage. Coordination service wires Master, Kernels, and Agents, although uses two types of transport – Master-Kernel interaction works on the top of ZooKeeper, interaction with Agents works on the top of CometD.

# Jagger Cluster :: 2



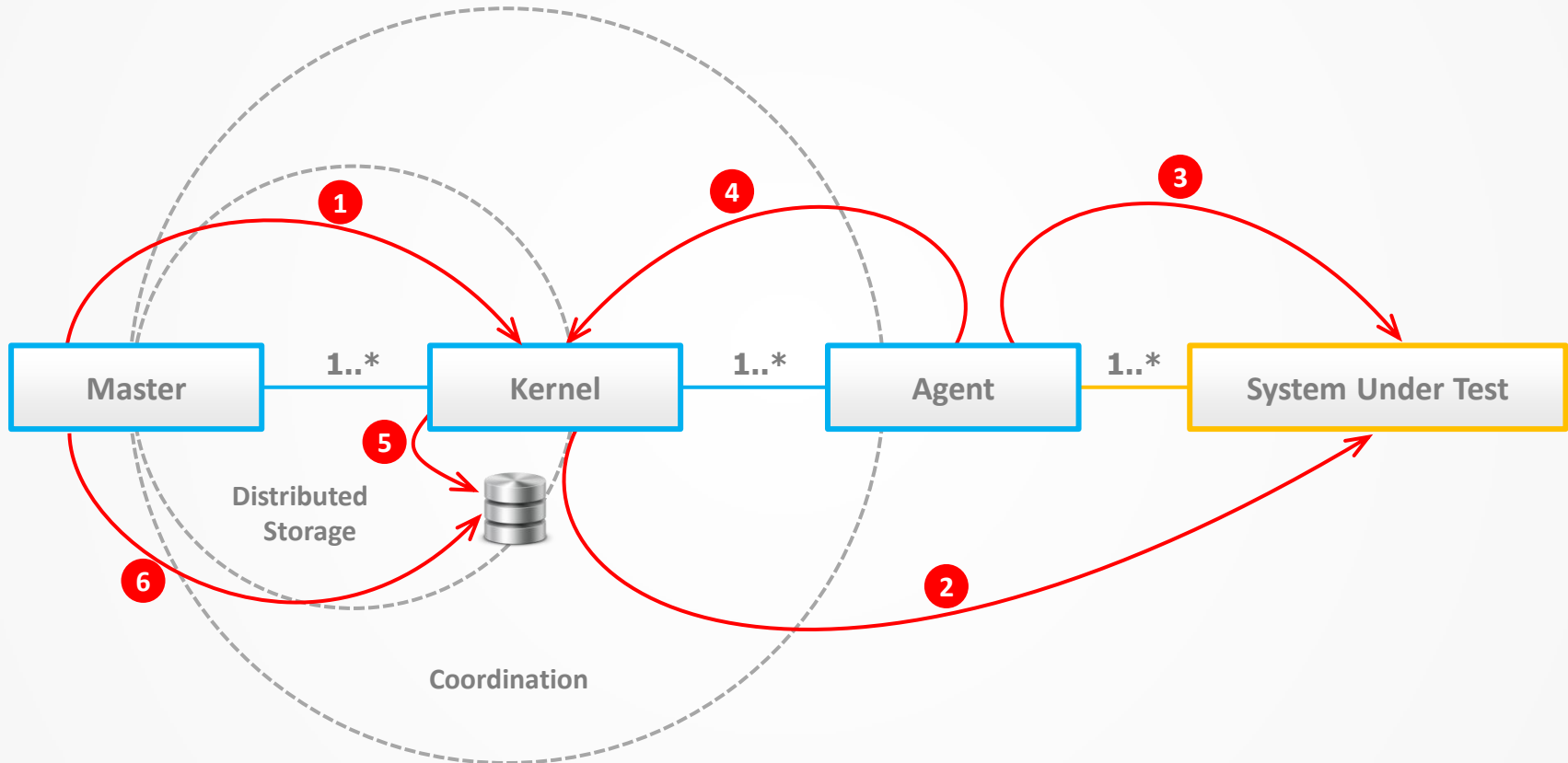
Let's consider cluster bootstrap routine. When Master starts, it waits for minimal number of Kernels which is specified in its configuration before test execution. Symmetrically, when Kernel is started, it continuously tries to find master. When both Master and Kernels are started they communicate one with other and Master register Kernel in its catalog.

# Jagger Cluster :: 3



Next step is registration of Agents. Each Agent first tries to find Master. As soon as Master become available, Agent request (1) address of the Kernel that will supervise it. Master replies with this address (2) and Agent register itself on the Kernel (3). At this step, cluster is ready to operate.

# Jagger Cluster :: 4



Test execution starts from the corresponding command from Master to Kernels (1) that contains test configuration. Kernel starts to workload the system under test (2), and request Agents to collect monitoring and profiling information (3). Agents continuously report collected information to the Kernel, and Kernel save both monitoring information and data collected during SuT invocation to the storage (5). Master continuously polls Kernels during this process and terminates test as soon as termination conditions are met. Finally, Master process data from storage (6) and produce a report.

# SATELLITE PROJECTS

# Jagger Projects

## Jagger



### Chassis

Core Jagger services,  
workload generation  
system



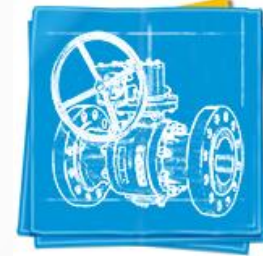
### Monitoring

Jagger distribution  
for monitoring



### Diagnostics

JVM Profiling library



### CloudShaper

A tool for system and  
network failures  
simulation

So far, Chassis, Monitoring, and partially Diagnostics systems have been discussed. Next slides provide a little bit more information about Diagnostics and CloudShaper. Both directions are very young and implementations are not mature. Jagger Team is looking forward for clients that are interested in these subprojects and ready to develop these directions.



# Jagger Diagnostics

- Jagger Diagnostics currently includes sampling profiler for JVM with automatic hot spots detection
- Jagger Diagnostics is an independent library that can be used to build profilers, production monitoring systems, and development tools

# CloudShaper

- CloudShaper is a tool that can dynamically adjust network bandwidth, simulate packets losses or nodes failures in distributed environments
- Technically, CloudShaper is a Groovy-based supervisor that issues commands via SSH according to some scenarios

# JAGGER R2 ROADMAP

# Jagger R2 Roadmap

- **Jaggerosity – Reporting and Management UI**
- **GroupBy feature in reporting**
- **Enhancements of Jagger SPI**
- **Performance Testing Server – remote API**
- **JVM memory profiling**

# Jaggerosity :: 1

Monitoring and Profiling yield huge amount of data. It is necessary to have interactive UI to browse this information efficiently.

The screenshot shows the Jaggerosity web application interface. The browser address bar displays "http://...". The application has two tabs: "Trends" and "Result Viewer", with "Result Viewer" being the active tab. On the left, a tree view shows the hierarchy of test results. The right pane contains a search filter and three tables of profiling data.

**Tree View Structure:**

- Session 001
  - Summary
  - Session Comparison
  - Test Summary
  - Test Details
    - Test T1
      - Monitoring
      - Profiling
        - Box 01
          - System 01
          - System 02 (highlighted)
        - Box 02
        - Box 03
        - Latency Plots
      - Test T2
      - Test T3
    - Session 002

**Search Filter:** filter

**Method Name Profiling Data:**

Method Name	On Top Ratio	In Trace Ratio
com.griddynamics....	12.23%	47.76%
org.apache....	8.90%	23.11%
org.apache....	5.90%	5.90%

**Callers:**

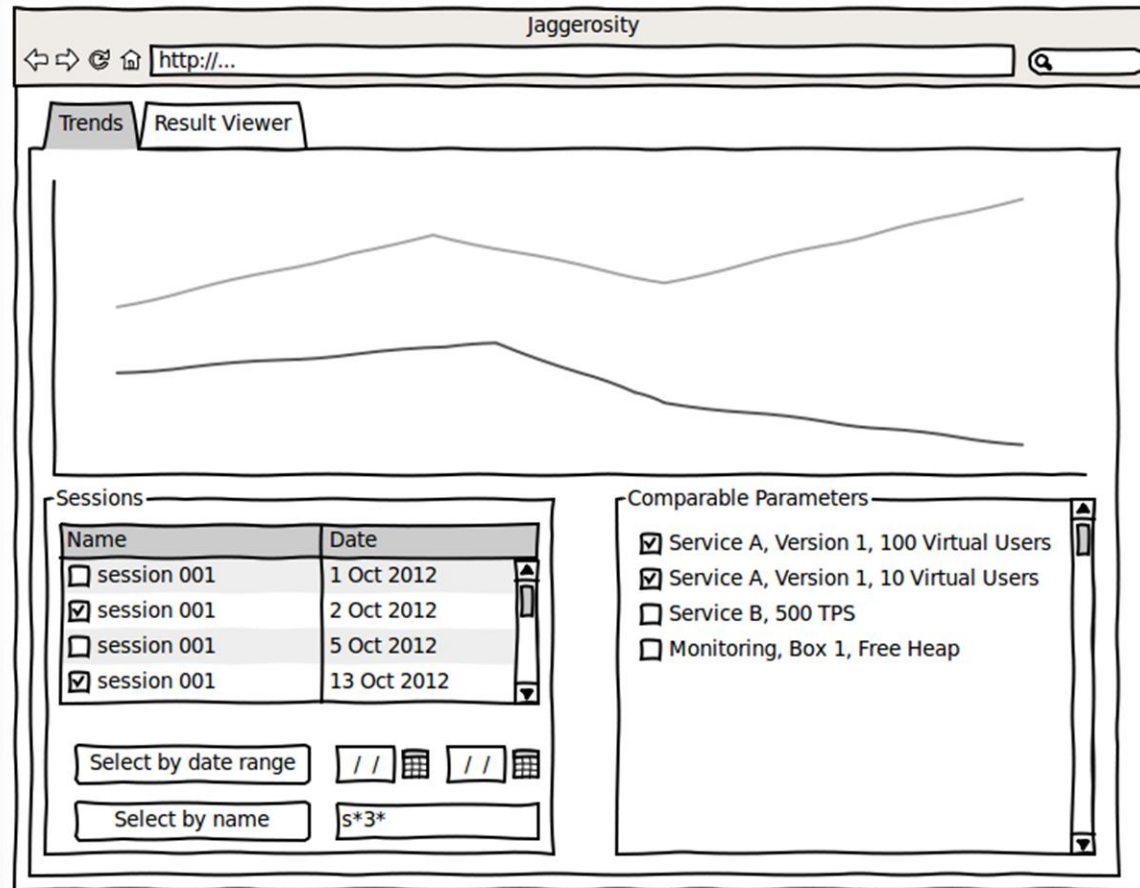
Method Name	On Top Ratio	In Trace Ratio
com.griddynamics....	12.23%	47.76%
org.apache....	8.90%	23.11%
org.apache....	5.90%	5.90%

**Callees:**

Method Name	On Top Ratio	In Trace Ratio
com.griddynamics....	12.23%	47.76%
org.apache....	8.90%	23.11%
org.apache....	5.90%	5.90%

# Jaggerosity :: 2

It is necessary to have interactive UI to compare historical data and visualize performance trends.



# GroupBy Feature

- Each transaction can be marked by System Under Test ID, query ID, custom transaction type (if test scenario contains multiple actions) etc.
- It should be possible to group measurements by such markers. For example, independent throughput/latency/monitoring plots and statistics for each System Under Test or query type.

JOIN JAGGER



# Join Jagger as a:

**Client** – try Jagger in your project, request a feature, share your experience in performance testing

**Architect** – review Jagger design, provide your feedback, device a new module

**Developer** – contribute to Jagger, participate in peer-to-peer code review

Contact Us

**[jagger@griddynamics.com](mailto:jagger@griddynamics.com)**

Distribution and Documentation

**<https://wiki.griddynamics.net/display/J4G/Home>**