Model-Driven Observability for Big Data Storage

John Klein
Software Engineering Institute
Carnegie Mellon Univ.
Pittsburgh, PA, USA
jklein@sei.cmu.edu

Ian Gorton
Northeastern University
Seattle, WA, USA
igorton@ccs.neu.edu

Laila Alhmoud, Joel Gao, Caglayan Gemici, Rajat Kapoor, Prasanth Nair, Varun Saravagi
Carnegie Mellon Univ.
Pittsburgh, PA, USA

***Abstract*—The scale, heterogeneity, and pace of evolution of the storage components in big data systems makes it impractical to manually insert monitoring code for observability metric collection and aggregation. In this paper we present an architecture that automates these metric collection processes, using a model-driven approach to configure a distributed runtime observability framework. We describe and evaluate an implementation of the architecture that collects and aggregates metrics for a big data system using heterogeneous NoSQL data stores. Our scalability tests demonstrate that the implementation can monitor 20 different metrics from 10,000 database nodes with a sampling interval of 20 seconds. Below this interval, we lose metrics due to the sustained write load required in the metrics database. This indicates that observability at scale must be able to support very high write loads in a metrics collection database.**

Keywords—big data, NoSQL, observability, model-driven engineering.

I. Introduction

In the last decade, the world has seen an exponential growth of digital data, Organizations such as Google and Facebook were born on the internet, and are leading this scale-driven revolution [1]. Beyond the Internet companies, big data applications are becoming pervasive across diverse business and scientific domains. For example, modern commercial airplanes produce approximately 0.5TB of operational data per flight [2], and by 2020, the Internet of Things (IoT) will generate 4 zettabytes of data per year, supporting monitoring and optimization of processes and services globally [3].

At the scale of these systems, meaningful analysis and prediction of end-to-end performance is usually not feasible at design time. Performance models must capture the complex static and dynamic component compositions in both the system and the underlying execution infrastructures. In addition, accurately representing heterogeneous and highly variable workloads challenges the state of the art in performance modeling. Pragmatically, even if it were possible to build such models, rapid post-deployment data growth, shared cloud-based infrastructures, and rapid application evolution would quickly invalidate model results. Assuring runtime performance at big data scale must be based on observing and analyzing *in vivo* application behavior. This enables *observability* into system health and status, both at the infrastructure and application level.

This paper builds on our earlier work [13] that presents the challenges of building massively scalable, easily configurable, lightweight observability solutions. In response to these challenges, we describe a model-driven framework for observability that is the focus of our current research. Model-driven approaches facilitate rapid customization of a framework and eliminate custom code for each deployment, hence reducing costs and effort. In our initial experiments, this framework has been able to efficiently collect and aggregate runtime performance metrics in a big data system with 1000s of storage nodes.

The contributions of our research in this area are:

1. A model-driven architecture, toolset, and runtime framework that allows a designer to describe a heterogeneous big data storage system as a model, and deploy the model automatically to configure an observability framework.
2. A reference implementation of the architecture, using the open source Eclipse package to implement the MDE design client, the open source collectd package to implement the metric collection component, and the open source Grafana package to implement the metrics aggregation and visualization component.
3. Performance and availability results from initial experiments, using the reference implementation.

The initial metamodel and implementation focuses on the pervasive big data pattern known as *polyglot persistence* [4], which uses multiple heterogeneous data stores (often NoSQL/NewSQL) within a single big data system. We note that a model-driven approach would not be strictly necessary (e.g., a discovery-based approach might be a better solution) if the observability scope was limited to just NoSQL/NewSQL technology. However, we intend this architecture to extend to cover complete big data applications, including processing pipelines and analytics. In this broader case, a model-driven approach provides advantages in automating and creating application-aware metric aggregation and visualization.

II. Architecture and Implementation

We have developed an architecture and a reference implementation[1] suitable for further research that addresses the challenges of observability in big data systems. The architecture uses model-driven engineering [5] to automate metric collection, aggregation, and visualization.

[1] Available at https://github.com/johnrklein/obs-prototype

978-1-5090-2131-4/16 $31.00 © 2016 IEEE
DOI 10.1109/WICSA.2016.27

A. *Overview of the Observability Architecture*

The architecture context is shown in Fig. 1, depicting three user roles. The first is *modeling*, representing a DevOps engineer who uses a design time client to create a model of the system's data storage topology and specify the configuration for each heterogeneous database. The model identifies the metrics to capture and their collection frequency. At the completion of the modeling phase, the design time client generates the monitoring configuration for a set of metric collection and visualization elements.

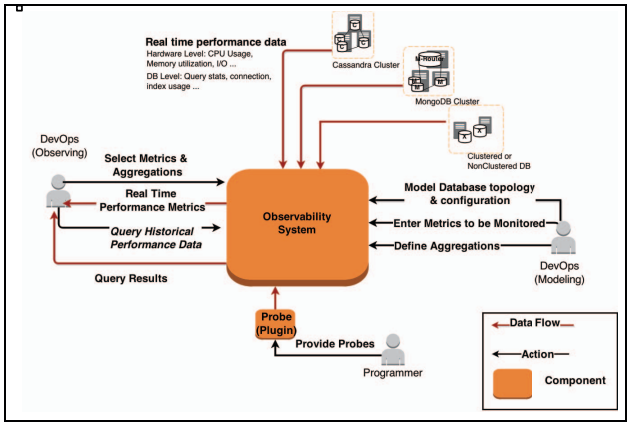

Fig. 1. Observability Architecture Context Diagram

The second role is *observing*, representing a system operator who uses a metric visualization client to monitor system performance. The visualization client is configured using the output of the design-time client to reflect the model of system to be observed. It supports real time monitoring of system operations and allows operators to specify and customize views based on their requirements for situational awareness.

The third role is *programmers*, who create metric collection probes to plug into the architecture. These probes are database-specific adapters that allow any database technology to be incorporated into the architecture. Extensibility is a major feature of our approach, as any solution must be able to efficiently support both existing and future database platforms.

The main run time elements of the observability system architecture are shown in the top-level component and connector diagram in Fig. 2. There are two clients, one for each of the main user roles, *modeling* and *observing*, discussed above. The *Server Tier* includes the *Metric Engine,* which implements dynamic metric aggregation and handles concerns related to dependability of connections to *Collection Daemons*. The *Server Tier* also includes the *Grafana Server*, which handles metric visualization. The *Model Handler* in the *Server Tier* propagates changes to the design-time model, and the *Notification Server* augments the interactive metric visualization with automated notification of user-defined exception conditions.

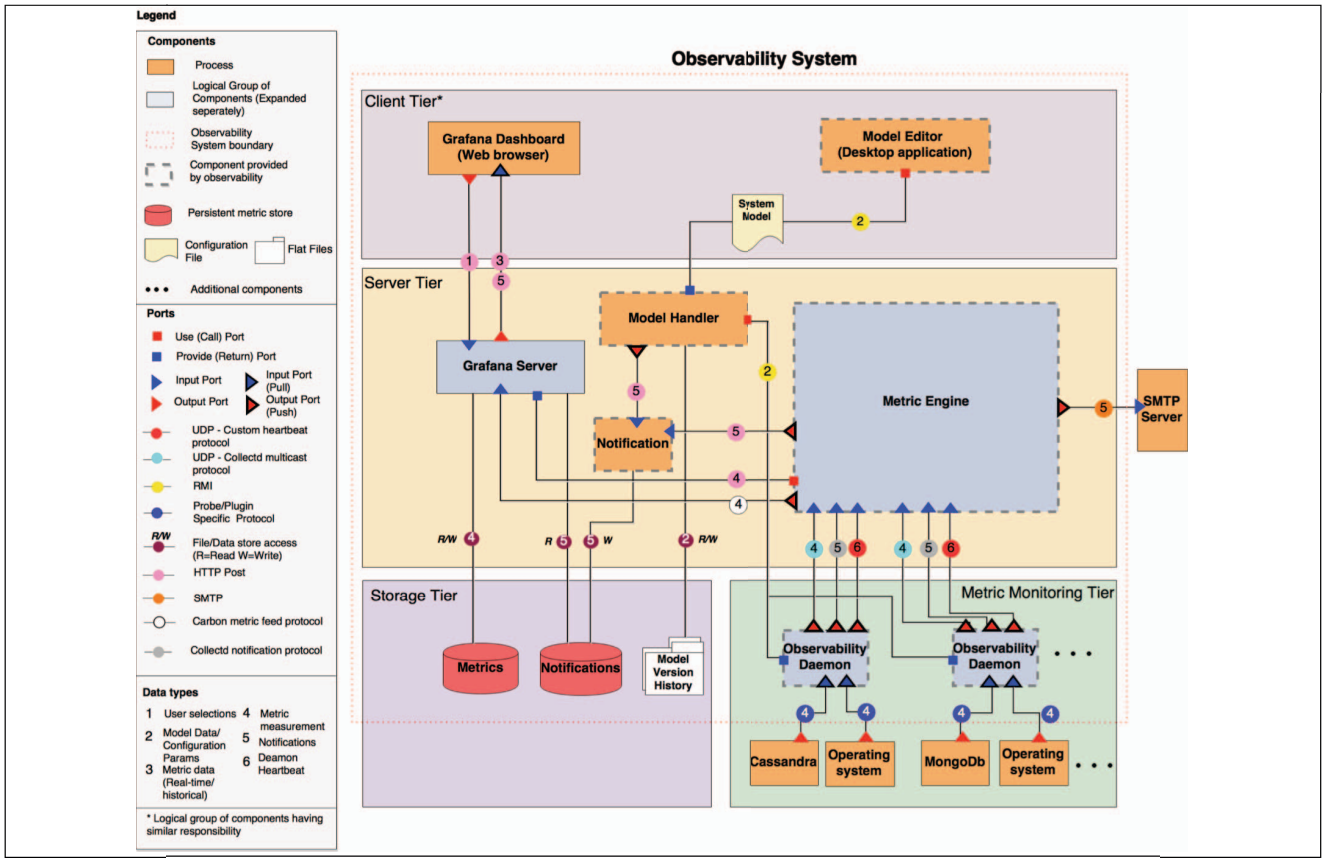

Fig. 2 Observability System Architecture (Component and Connector View)

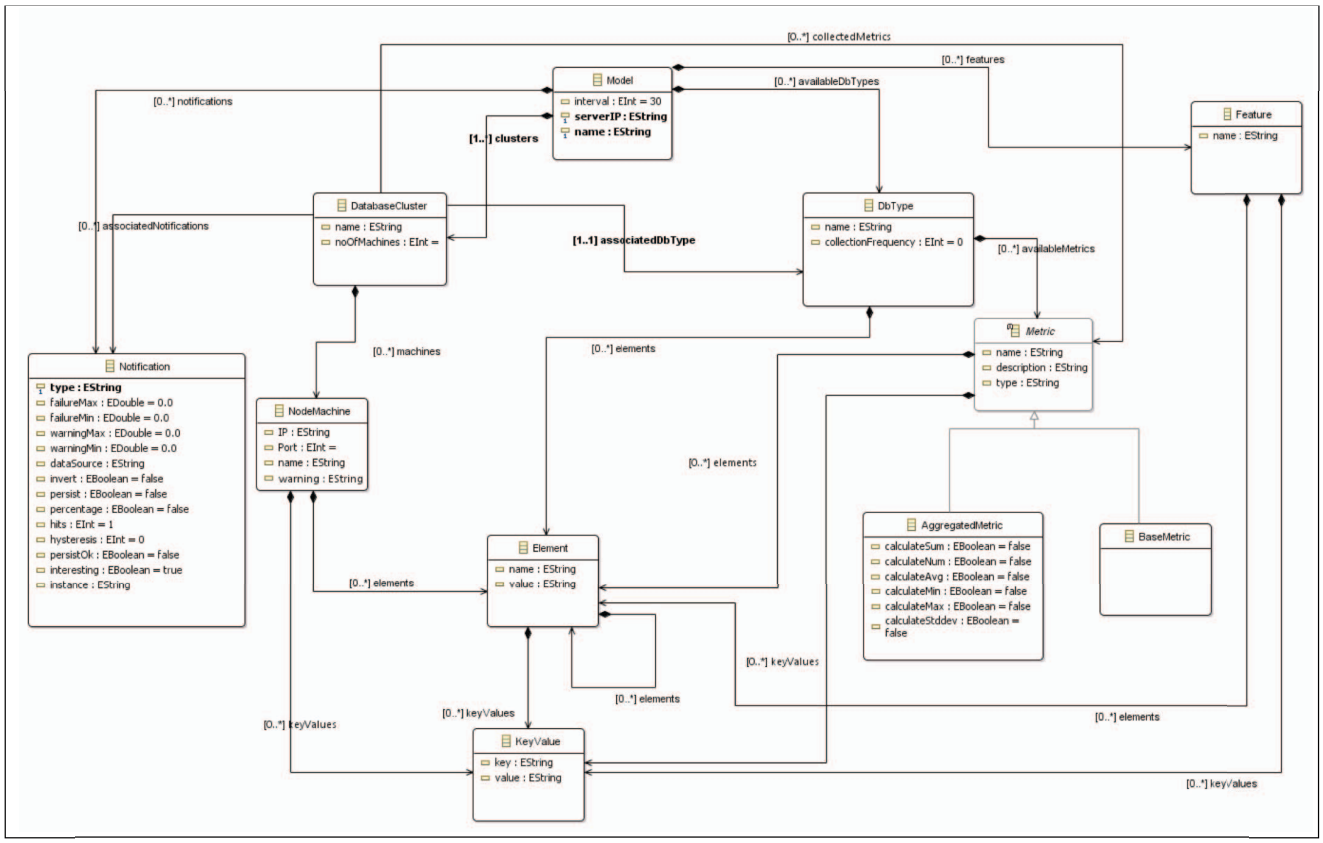

Fig. 3. Metamodel for observability of polyglot persistence pattern

The *Storage Tier* provides persistent storage of metric streams and notifications. All metrics for each database are stored as a time series to facilitate visualization and analysis. Metrics are stored with metadata to enable dynamic discovery of the metrics. This is necessary to accommodate changes in monitoring configurations after an existing model has been upgraded and deployed as a new version.

The *Metric Monitoring Tier* uses *Observability Daemons* on each database node to collect metrics from the local database instance and operating system. The daemons exploit database-specific and operating system APIs to periodically sample metrics and forward these to the *Metric Engine.*

B. Metamodel

Our observability architecture exploits a model-driven engineering approach to address the scale challenge of big data systems. Hence, a model of the system to be observed is created by the *modeling role*. This model is built from elements defined in the metamodel (Fig. 3). It specifies and customizes the components in our observability framework. The system model also specifies the metrics to be collected and how the metrics will be aggregated. The metamodel represents the topology as one or more database clusters (*DatabaseCluster* element in the metamodel), with each cluster using a particular technology (*DbType*). A cluster is comprised of a number of nodes (*NodeMachine*).

Metrics (*Metric*) are defined as key-value pairs (*KeyValue*) collected from a database cluster. They may be simple values collected directly from a database's monitoring API (*BaseMetric*), or calculated at run time from one or more simple metrics (*AggregatedMetric*). Each particular database technology (*DbType*) defines a set of metrics that are supported by that technology and are available programmatically. This approach makes it possible to collect both common metrics that are available from all databases (e.g., disk utilization, query processing time) and technology-specific metrics (e.g., automatic rebalancing in MongoDB [2]). These metrics are available for selection when the modeler creates a system model and specifies the databases that will be deployed.

The metamodel also defines the structure of event notifications (*Notification*). These are triggered when simple or aggregated metrics exceed a specified threshold value set by the modeler.

C. Model Editor Client

In Fig. 2, the *Model Editor Client* instantiates the metamodel in a graphical editor, using the Eclipse Modeling Framework (EMF) [3]. The graphical model specifying the topology of the observed system and the metrics to be collected

[2] https://docs.mongodb.org/manual/core/sharding-balancing/

[3] https://eclipse.org/modeling/emf/

and aggregated is transformed using Acceleo[4] into a text representation. This is uploaded to the *Model Handler* server, which propagates model changes to the *Observability Daemons* and *Metrics Engine*. Models are versioned to improve dependability, ensuring all parts of the system are consistent. This also enables rollback to a previous version.

D. Runtime Metric Collection

An *Observability Daemon* executes on each node in the observed system to collect metrics and forward them to the *Metrics Engine*. Each *Observability Daemon* is configured by the *Model Handler* based on the system model, so that model changes (e.g., in topology or metrics collected) immediately change the *Observability Daemon* behavior. The *Observability Daemon* is based on the collectd[5] open source package. Our architecture adds a *Daemon Manager* component on top of collectd, so that collectd can be remotely and dynamically configured by the *Model Handler*.

Our architecture uses collectd plug-ins to adapt to each supported database technology, encapsulating the precise mechanism used to obtain database metrics within a database-specific plug-in. The plug-ins exploit the monitoring API provided by the specific database technology (e.g., Cassandra's JMX API[6]), to acquire the metric data from databases. We have developed and tested reference plug-ins for Cassandra, MongoDB, and Riak.

Each *Observability Daemon* sends the collected metrics to the *Metric Engine* server using the collectd notification protocol. Separately, a heartbeat notification is sent by the *Daemon Manager* to the *Missing Daemon* component in the *Metrics Engine*. The *Missing Daemon* component uses the system model to determine which *Observability Daemons* should be executing, compares that to the received heartbeats, and raises an alarm when an *Observability Daemon* appears to have failed. The heartbeat notification was included to improve dependability. Simply monitoring a metric stream to assess the state of an *Observability Daemon* is problematic as individual messages may be delayed due to transient network partitions, or daemon or node failure. Transient partitions can be handled by this protocol, as daemons buffer collected metrics for a configurable time period (e.g. 5 minutes) and can resend missed values. Node and daemon failures currently require operator intervention. Automated recovery from such failures simply requires us to incorporate additional monitoring capabilities into our framework, an objective for further work

E. Metric Aggregation and Visualization

In our implementation, several metric visualizations are created using the Grafana[7] open source package, which supports time series graphs such as those shown in Fig. 4. Grafana comprises a server and a web-based client, as shown in Fig. 2. Metrics are stored using self-describing data structures embedded in the metric stream. We utilize key-value pairs, where the metric name comprises the key and the value is the metric reading at a given time.

Fig. 4. Metric visualization user interface

III. Performance Results

To assess the performance and scalability of our observability architecture, we performed a series of tests using Amazon's AWS cloud platform. We created a test daemon that was able to simulate metrics generation from multiple database nodes. We then configured a pool of test daemons to simulate metrics collection from 100 to 10,000 database nodes. We initially set the metrics collection interval to 30 seconds, and configured the daemons to simulate the generation of 20 distinct metrics per node. We also specified the model to aggregate CPU metrics from all nodes to calculate overall system CPU utilization.

We deployed the observability architecture on an AWS m3.2xlarge instance type. This comprised an Intel Xeon E5-2670 v2 (Ivy Bridge) server with 8 cores, 8MB RAM, 30GB disk and 160GB SSD. The test daemons were configured to initially simulate metrics generation from 100 database nodes. After 5 minutes, the number of simulated nodes increased to 1000, and then 1000 simulated nodes were added every 5 minutes until the test simulates 10,000 database nodes. We monitored the resource usage on the observability server, and the results are in Fig. 5.

Graph 1 shows that the metric transfer time stays constant as the number of metrics per interval increases (the peaks in this graph), and that the metric transfer takes about one-half of the collection interval, leaving margin for growth (the troughs in this graph). The system was able to handle 10,000 nodes generating 20 metrics during each 30 second interval. The system scaled well to handle network traffic and saved the metrics to disk to be shown at dashboard. The aggregation plugin was able to aggregate metrics from 10,000 nodes successfully.

To summarize the test results:

- With linearly increasing metric collection load, the disk space used also increased linearly. For 10,000 nodes with 20 metrics per node and a 30 second collection interval, disk space required is approximately 50 GB for 7 days monitoring data.

[4] http://www.eclipse.org/acceleo/

[5] https://collectd.org

[6] http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsMonitoring.html

[7] http://grafana.org

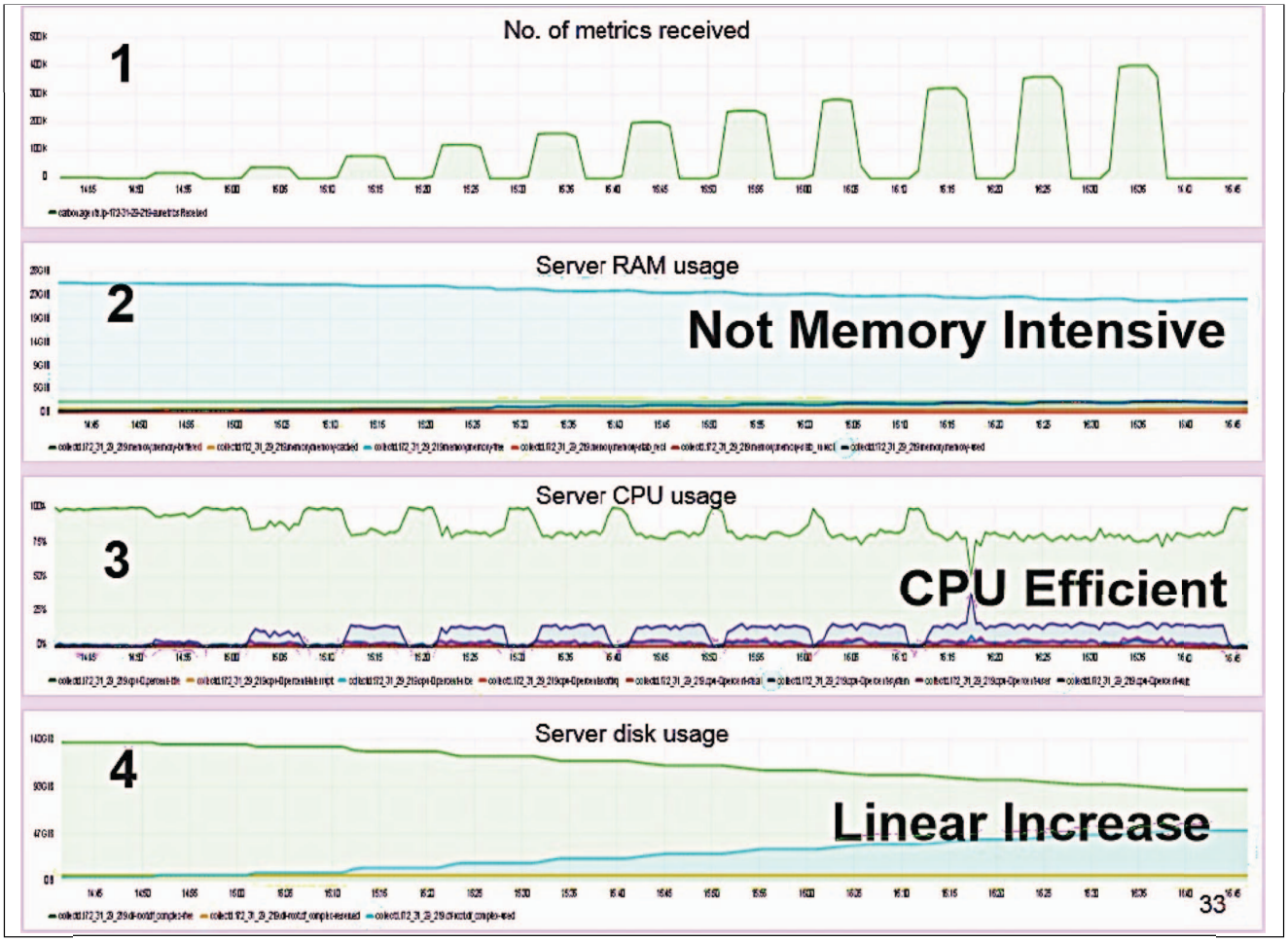

Fig. 5. Performance and Scalability Test Results

- Server free memory reduced from 25 GB initially to 22 GB with 10000 nodes being monitored. Hence we conclude our solution is not limited in scalability by memory utilization.
- CPU utilization is low, only showing minor increases in activity as the number of nodes grows.
- With 10000 simulated nodes, the server was processing 293 Kbits/s of network traffic at peak.

To stress test the observability framework, we deployed the test system with 10,000 simulated nodes. The collection interval started at 30 seconds, and every 5 minutes was reduced by 5 seconds. The system operated normally until the sample frequency reached 15 seconds. At this point, some metrics were not written to disk. This situation continued to deteriorate as we reduced the sampling interval to 5 seconds. No components failed, but there was a significant loss of metric data in the database.

Examining execution traces from these failing tests, we saw the CPU, memory, and network utilization levels remained low. Disk writes, however, grew to a peak of 32.7 MB/s. This leads us to believe that the Whisper database[8] in the Grafana server was unable to sustain this relatively heavy write load. This is likely a limitation of this component in our architecture. Replacing this database with a distributed database technology such as Cassandra would consequently make it possible to monitor a significantly larger collection of nodes.

IV. Prior Work

There has been significant prior work on collecting general measurements of resource utilization at process and node level. This includes Ganglia[9], and Nagios [6]. Ganglia and Splunk[10] support collection of host-level measurements across clusters, and provide basic monitoring and visualization dashboards. Commercial products from HP[11], IBM[12], and others also provide similar collection and visualization capabilities, but

[8] http://graphite.wikidot.com/whisper

[9] ganglia.sourceforge.net

[10] www.splunk.com

[11] http://www8.hp.com/us/en/software-solutions/systems-management-server-monitoring-tools/

[12] http://www.ibm.com/software/tivoli

incur significant license costs. Tools such as Chukwa[13] and Sawzall[14] focus on general analytics on collected log data, and provide semantics for time series data sets.

In visualizing large-scale system health and performance, Yin, et al, take an approach inspired by video games to enable navigation through a complex data landscape [7]. In this case, the focus was on infrastructure-level measurement data, however the approach may be extensible for other types of measurements. The Theia system provides architecture-specific visualization for Hadoop-based systems [8].

Architecture-aware modeling based on architecture styles traces back to very early work in software architecture [9]. More recently, Palladio [10] uses architectural styles to generate performance models, and Rainbow [11] uses architectural styles to model and generate a runtime framework focused on dynamic adaptation. The Rainbow framework uses measurement probes, which may include monitoring performance. However, the probes must be built into the components of the system, and the generation focuses on style-based reaction strategies when a probe's measurement crosses a threshold.

Finally, there has been little work on using model-driven approaches to generate monitors. He, et al, present a model-driven approach to composing monitors, synthesizing a compatible metamodel and then transforming heterogeneous monitors into that common metamodel. The approach generates only monitors, without aggregations, a measurement persistence schema, or visualizations [12].

V. Conclusions and Future Work

In this paper we described the design and prototype implementation of an observability framework for big data systems. We have exploited model-driven techniques to make the core architecture customizable to different system's observability requirements without the need for custom code for each deployment. We have also built the solution by reusing various off-the-shelf components to streamline our development effort and provide advanced capabilities 'out of the box'. The reference implementation has been publicly released as a research platform. The reference implementation has availability limitations that will be addressed, using standard architecture mechanisms, as we evolve the platform.

Our current implementation only provides observability at the database layer. Extending these model-driven capabilities to other layers in a big data system (e.g., application server and Web servers) and improving the scalability of our framework forms the core of our future work. We also wish to investigate the potential of automated resource discovery approaches to compose an observability system dynamically. Automated approaches have immense potential for dealing with scale and rapid evolution, but face many daunting challenges, for example navigating security perimeters, distributed data centers and logical application partitions.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003210

References

[1] J. Weiner and N. Bronson. *Facebook's Top Open Data Problems* [Online]. https://research.facebook.com/blog/1522692927972019/facebook-s-top-open-data-problems/ (Accessed 10 Nov 2014).

[2] M. Finnegan, "Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic," *Computerworld UK,* 6 March 2013, http://www.computerworlduk.com/news/infrastructure/3433595/boeing-787s-to-create-half-a-terabyte-of-data-per-flight-says-virgin-atlantic/ (Accessed 20 Feb 2014).

[3] V. Turner, J. F. Gantz, D. Reinsel, et al., "The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things." International Data Corporation, White Paper, IDC_1672, 2014, http://idcdocserv.com/1678 (Accessed 10 Nov 2014).

[4] P. J. Sadalage and M. Fowler, *NoSQL Distilled.* Addison-Wesley Professional, 2012.

[5] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice.* Morgan & Claypool, 2012.

[6] E. Imamagic and D. Dobrenic, "Grid Infrastructure Monitoring System Based on Nagios," in *Proc. 2007 Workshop on Grid Monitoring (GMW '07)*, Monterey, California, USA, 2007, pp. 23--28. doi: 10.1145/1272680.1272685.

[7] J. Yin, P. Sun, Y. Wen, et al., "Cloud3DView: An Interactive Tool for Cloud Data Center Operations," in *Proc. ACM Conference on SIGCOMM (SIGCOMM '13)*, Hong Kong, China, 2013, pp. 499--500. doi: 10.1145/2486001.2491704

[8] E. Garduno, S. P. Kavulya, J. Tan, et al., "Theia: Visual Signatures for Problem Diagnosis in Large Hadoop Clusters," in *Proc. 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques (lisa'12)*, San Diego, CA, 2012, pp. 33--42.

[9] M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[10] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *J. of Systems and Software,* vol. 82, no. 1, pp. 3-22, Jan 2009, doi: 10.1016/j.jss.2008.03.066

[11] D. Garlan, S.-W. Cheng, A.-C. Huang, et al., "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure," *IEEE Computer,* vol. 37, no. 10, October 2004, doi: 10.1109/MC.2004.175.

[12] Y. He, X. Chen, and G. Lin, "Composition of Monitoring Components for On-demand Construction of Runtime Model Based on Model Synthesis," in *Proc. 5th Asia-Pacific Symposium on Internetware (Internetware '13)*, Changsha, China, 2013, pp. 20:1--20:4. doi: 10.1145/2532443.2532472

[13] John Klein and Ian Gorton. 2015. Runtime Performance Challenges in Big Data Systems. In Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development (WOSP '15). ACM, New York, NY, USA, 17-22.

[13] wiki.apache.org/hadoop/Chukwa

[14] research.google.com/archive/sawzall.html