Learning to Simplify Distributed Systems Management

Christopher Streiffer [*], Ramya Raghavendra [†], Theophilus Benson[‡], Mudhakar Srivatsa[†]
[*]*Duke University, Durham, NC, USA. Email:* `cdstreif@cs.duke.edu`
[†]*IBM T.J. Watson Research Center, Yorktown, NY, USA. Email:* `{rraghav,msrivats}@us.ibm.com`
[‡]*Brown University, Providence, RI, USA. Email:* `theophilus_benson@brown.edu`

***Abstract*—Managing large-scale distributed systems is a difficult task. System administrators are responsible for the upkeep and maintenance of numerous components with complex dependencies. With the shift to microservices-based architectures, these systems can consist of 100s to 1000s of interconnected nodes. To combat this difficulty, administrators rely on analyzing logs and metrics collected from the different services. However, the number of available metrics for large systems presents complexity and scaling issues. To combat these issues, we present *Minerva*, an unsupervised Machine Learning (ML) framework for performing network diagnosis analysis. *Minerva* is composed of a multi-stage pipeline, where each component can act individually or cohesively to perform various management tasks. Our system offers a unified and extensible framework for managing the complexity of large networks, and presents administrators with a swiss-army knife for diagnosing the overall health of their systems. To demonstrate the feasibility of *Minerva*, we evaluate its performance on a production-scale system. We present use cases for the various management tools made available by *Minerva*, and show how these tools can be used to make strong inferences about the system using unsupervised techniques.**

I. Introduction

Distributed systems are constantly monitored to understand their current state and reason about future performance. Metrics enable application developers and system administrators to gain *actionable insights* that can be meaningfully utilized for common management tasks such as resource allocation, performance insights, anomaly detection, and debugging failures.

Cloud operators provide monitoring infrastructure for application developers, for example, Amazon Web Services' CloudWatch [9], Google's Stackdriver [20], and Azure's Monitor [27]. Metrics specific to the application as well as systems and networking can be obtained for a price. While there has been focus on scalable data collection and storage, the task of reasoning about the data and making actionable conclusions is left to the application developers.

As applications get more complex, and levels of hardware and software abstraction increase, the community is increasingly adopting the "collect everything" approach to monitoring. This approach adds an increased burden on the 'experts' who are forced to make sense of the application dependencies, identify groups of equivalent signals, and frame the right queries that may lead to problem diagnosis. On talking to developers and architects, we found this to be a huge pain point with a real need for added intelligence and analytics to ease this burden.

In this work, we ask the question *can cloud management be unsupervised?* Ideally, we would like for the metrics to point us to what the potential problems could be, instead of the current method of an expert sifting through vast amounts of data using domain expertise, intuition and experience. Our work builds on a confluence of trends – on one hand, advances in deep learning techniques provide high accuracy but require an immense amount of data, while on the other hand, the growing adoption of microservices-based architectures generate an immense amount of data. Together, these two trends allow us to apply machine learning, and specifically deep learning to cloud management. Given the advances in machine learning algorithms and hardware on the one hand, and the popularity of microservices based application architecture on the other hand, we believe that the time is right to use one to aid the development of the other.

After several discussions with cloud application developers, we distilled their set of concerns to a few major ones: *Can we forecast the performance of individual services as well as the entire system? Can we detect performance anomaly or degradation? Finally, can we analyze the root cause of the problem?*

Performance **prediction** [34], [30], [18], [25] is a key management primitive for various tasks including resource allocation and traffic engineering. Yet in the presence of complex and high dimensional dependencies of a microservices-based architecture, it becomes intractable to make accurate predictions or determine the root cause of performance issues. As a result, **dimensionality reduction** [38], [32], [33], [12], [26], [15] is a key primitive provided by our system that assists in several management tasks - clustering services that exhibit similar temporal behavior for system interpretability, or detecting anomalies when a service's temporal fingerprint changes. Additionally, **causality extraction** [10], [35], [7], [14], [6] provides insight into service relationships which not only assists in localizing faults, but leverages dependencies to improve predictions.

While a rich body of prior work on distributed systems and cloud management exists in literature [35], [10], [14], [6], we found two major drawbacks. First, existing systems are limited in that they are point solutions that perform a limited scope of management. Our vision is to build a unified framework, ideally one that can be extended to assist all management needs. Second, we want to reduce reliance on experts or supervision, and instead draw from the advances in deep learning techniques with scalable feature extraction for accurate and actionable insights.

To that end, we build *Minerva*, a Machine Learning (ML) framework that uses unsupervised learning to improve the effectiveness of cloud management by (1) improving scalability by reducing metric dimensionality e.g. clustering, (2) improving efficiency by removing duplicate signals for root cause analysis (RCA) e.g. dependency detection and aggregation (3) applying deep learning techniques on the data, and leveraging the inferences from the previous steps to provide accurate prediction.

We posit this work as a first step in the direction of organizing metrics and discovering the dependencies between which are critical cloud management tasks. There are several promising future directions that are opened up including automated root cause analysis, augmenting expert knowledge to automated analyses that we plan to pursue in the future.

II. Background

In this work, we focus on microservices-based distributed systems, wherein loosely coupled distributed components communicate via well-defined interfaces. Applications built in this manner are typically composed of 100s of components. This results in increased application complexity, and an explosion in the number of metrics that are collected for the purposes of debugging, monitoring performance and application management. Not only does it become difficult to understand how the application performs as a whole, but when performance issues arise, analyzing the dependencies between components and using these dependencies along with the metrics becomes extremely challenging.

A combination of metrics is known to be significantly better predictor of service level objective violations than individual metrics [40]. A typical approach to managing large scale metrics data is to use a data processing pipeline with a collection engine such as logstash [4] or statsd [16], a database [1], [5], followed by a dashboard such as Kibana [3] or Grafana [21]. The final step however still requires for an ”expert” to step in and troubleshoot when there is a performance issue by framing the right queries, searching for a needle in the haystack. Many in-house alternatives to open source components are developed in order to allow for systems to scale to the high metrics volumes [29], [6], [37]. These tools are developed to be application specific, however, and require either application instrumentation or sophisticated techniques to infer causal relationships by analyzing message trace timestamps, making them inapplicable for broader use.

Major cloud computing operators [9], [8], [27] also provide monitoring tools for recording all metric data from all components. These tools, however, either rely on few system metrics that are hand-picked via experience, or recording all metric data for all components, because they do not know how to filter redundant and less useful metrics beforehand. For example, Uber reports of collecting 500M timeseries [36], and Netflix similarly reports scaling their systems to 20M metrics [6].

Relying on past experience and default metrics may not always be beneficial due to the increasing complexity of a microservices-based application. On the other hand, recording all metric data can create monitoring overhead in the network and storage, and in the case of applications being run on a cloud platform, it can add to costs where charging is done on a per-metric basis [9]. For these reasons, it is important to help the developers identify and minimize the critical components and metrics to monitor, and uncover dependencies between the components of a microservice-based application. Ideally, this process should be generalizable and not be intrusive to the application.

III. Related Works

Our work is motivated by the large body of existing work for managing distributed systems at scale at scale [35], [10], [14], [6], and their limitations.

System Monitoring: System architects utilize distributed tracing systems such as Dapper [35], X-Trace [17], Pinpoint [13], and Magpie [11] to monitor the performance of their systems. These services manage the collection, lookup, and visualization of data aiding developers in debugging. However, these services also require code instrumentation which is often limited in coverage and imposes an execution overhead on the applications running in the system.

Facebook's Mystery Machine [14] is designed to not require instrumentation, instead relying on log messaging with details of the request ID, executing computer, etc. to perform its analysis. Bahl et al. [10] present Sherlock, a framework for performing RCA for failures within an enterprise-level distributed system. Their framework requires special software agents running on each node and is designed to perform failure localization.

Minerva is designed to be minimally invasive by performing all of its analysis on metrics collected from system services using unsupervised techniques, rather than relying on instrumentation or specialized software running on each node. Further, should tracing information be available, *Minerva* can improve its performance through the use of semi-supervised machine learning techniques.

Metric Aggregation and Visualization: Netflix's Vector [29], Amazon's CloudWatch [9], Google's Stackdriver [20], and Microsoft's Monitor [27] all provide powerful logging, monitoring, and visualization tools to create a comprehensive system overview. Amazon offers an AutoScaling [8] platform which integrates with CloudWatch to automatically scale up or down system components based on alerts created by the user. *Minerva* provides a powerful tool for filtering and isolating metrics to select for monitoring, and can be integrated with these services to present a more succinct system visualization.

ML for Systems: Several machine learning methodologies exist for clustering time series data [32], [33], [38], which can be used to perform dimensionality reduction. Existing monitoring systems [12], [26], [15] use clustering to perform a dimensionality reduction, but do not make any causal inferences about the clusters, nor use the clusters for any advanced statistical inferences e.g. regression. *Minerva* takes advantage of clustering methodology to make inferences about related services in order to reduce redundant metrics, and uses these results to improve the performance of causal analysis and regression.

CloudScale [34], AGILE [30], and PRESS [18] use statistical and signal processing methodologies to predict resource usage in order to perform dynamic auto scaling to meet service level objectives (SLOs) while maintaining a low resource provisioning cost. Mao et al. [25] use a deep reinforcement learning framework for managing resource scheduling in large-scale systems. Google's DeepMind [19] uses machine learning to operate their data center networks more efficiently, and has been successful at predicting Power Usage Effectiveness (PUE) to reduce energy costs.

Minerva takes a more generalized approach by identifying significant metrics, which compliments these methodologies by figuring out which metrics are useful for specific applications.

Our work sets to ask the question what are the core primitives we need to provide to cover such a rich body of work, while at the same time be flexible enough to incorporate advances in the field? While machine learning has been demonstrated useful in fault detection, we would like to propose a unification of such point solutions to build a non-intrusive system with low overheads, which uses unsupervised techniques to make strong inferences about the underlying system.

IV. System Design

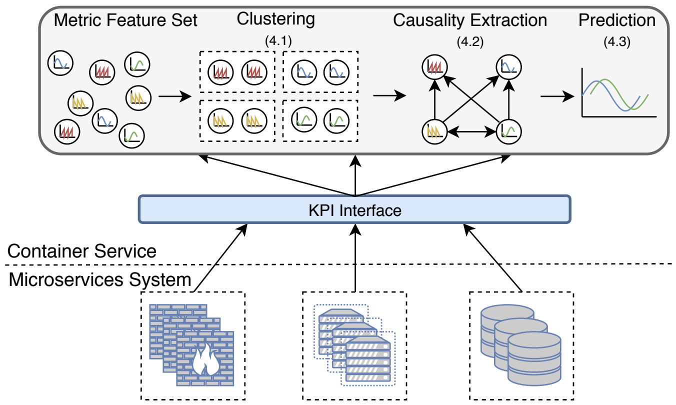

Figure 1: This figure shows the design of the system in a production environment. Each node or service within the system reports its metrics (interchangably used with KPI metrics) to the database. *Minerva* interacts with the different components of the system through a KPI time series database and runs its analysis.

Minerva aims to tackle cloud management by providing a principled framework for performing RCA (Root Cause Analysis), anomaly detection and prediction in an efficient and mostly unsupervised manner. Central to *Minerva*'s design is the insight that efficient unsupervised learning requires a combination of techniques for reducing noise and amplifying signal. To this end, *Minerva* encompasses a chain of interdependent learning modules, presented in Figure 1, which interact through clean and well defined interfaces. The three modules that constitute *Minerva* include: *dimensionality reduction* – which groups or clusters similar entities; *noise cancellation* – which captures dependencies and aggregates entities to enrich the feature set; and, *prediction* – which analyzes the resulting feature set to detect anomalies and predict resource utilization.

At a high level, *Minerva* is a sequential ordering of three abstractly defined-components where the output of each component can be used for network management tasks independently, or fed as input into the next stage. This flexible design allows for *Minerva* to adapt to different system and network needs. Each component is designed as an abstraction, and there exists an API between the adjacent components for simplified communication.

We now describe the different components of *Minerva*, their input and output interfaces, and how they interact.

A. Dimensionality Reduction: Clustering

Consider the question of which of all the logged metrics are "relevant" for a given application. Answering this will aid in understanding application performance, orchestration and debugging. To address this problem, the first component of *Minerva* is responsible for reducing the number of metrics to be collected, and grouping them into similar ones to be processed by subsequent components. At a high level, this component is a light weight technique for filtering out unrelated time series and clustering similar ones. Very large metric space implies some of the computationally heavy downstream components are infeasible to run; *Minerva* allows these components to run in real time. If the subsequent components have an abundance of computation resources or the number of available time series is small, the components can choose to operate on the raw data. However, from our experience with systems, even a moderate sized system with few hundred time series blows up the time required for downstream analytics. Hence this steps provides a critical preprocessing step; subsequent components can in turn process centroid time series — a representative time series from each group.

The purpose of this methodology is to discover nodes and services which display similar usage patterns. The input to this component is the time-series data collected from each node or service running within the system. Optionally, this component additionally accepts as input "hints" about the potential clusters. For example, in a service-level analysis, the user could provide the label "hadoop-container" to indicate which metrics should be mapped to the same type of service. Because it is assumed that these services perform the same task, it is a reasonable assumption to make that their metrics will end up in the same cluster. When the node-labels are included, the metrics will be passed through a preprocessing phase to compute the initial centroids. The output of this component is the set of clusters which contain mappings to the input metrics and the corresponding centroid of each cluster.

Interaction with Next Stage: Having this component first allows for *Minerva* to perform a dimensionality reduction on the total number of metrics. As mentioned previously, some systems can have over 1,000,000 metrics making the downstream components computationally infeasible to run. This step can be summarized as performing a correlation analysis between the different nodes/services, allowing for the dimensionality of the input feature set to be reduced.

Design Requirement: While *Minerva*, does not prescribe the set of algorithms used in any of the components, *Minerva* does present requirements for each of these methodologies. For the dimensionality reduction component, the algorithm must be able to do (1) effectively and reliably group time series with minimal user effort while being able to tolerate different amplitudes and shifts in time – given two different attempts to group the same dataset, the component must arrive at the same groups, and (2) for each group, the algorithm must produce a representative time series. This time series can either be a member of the group or a new derivative time series created as a function of members of the group e.g. the centroid of the cluster.

Management Use Case: *Metrics reduction* is a key use case for this step. Manually investigating metrics from large number of components becomes exceedingly difficult with increasing number of metrics and components, as such, not all metrics that are strongly correlated with each other are necessary to make decisions about the control of an application. Further, public cloud providers such as Amazon CloudWatch charge on a per-metric basis, so metrics reduction helps in *cost reduction*. Clustering allows for the different services and nodes operating within the system to be *recovered*. What this means is that assuming none of the labels were known for the metrics, it is possible to reconstruct which metrics belong to which service type. This has the direct effect of constructing a fingerprint for the various services running within the system to enable performing anomaly detection as each subsequent time series can be compared against the centroids of the results.

B. Noise Cancellation: Causality Extraction

Consider the problem of automatically inferring complex application dependencies in order to provide actionable insights into system performance. The second component of *Minerva* is responsible for performing a causal extraction on the different time series (or centroids) and to aggregate dependent time series to eliminate duplicate and spurious alerts. Whereas the first component reduces data to be processed by subsequent ML algorithms, this component focuses on reducing the amount of information presented to the operator. In essence, extracting causality and grouping dependent time series enables *Minerva* to minimize "alert fatigue" — a situation in which diagnosis is impeded by operators facing too many alerts. An added benefit of aggregating dependent time series is that this leads to an amplification of the problem, further enabling better detection by the subsequent component.

Design Requirement: For inferring dependencies, the system and algorithm must be capable of (1) effectively and reliably recognizing causal relationships between services by observing directly measurable cause-effect factors such as metrics data, tracing, call logs etc.; (2) adapting to changes by updating the causal graph in real time to reflect the current system state. The input to this component is either the time-series data from the entire system, or if this input set is too large, the centroids from the metric clusters computed from the previous component. The output of this method is a reconstructed causal graph between the different metrics/nodes/services (can be selected between).

Interaction with Next Stage: Because the next stage performs prediction, the predictive capabilities of the model can be strengthened by knowing which metrics causally influence other metrics. This allows for these metrics to be grouped together to create a richer feature set.

Management Use Case: Finding causal nodes and metrics within the network can be used to perform *RCA* and *anomaly detection*. For the RCA identifying upstream causal nodes which impacts "faulty" downstream nodes can save significant time and debugging effort by pointing users in the direction of where to look. Auto-scaling [8] decisions can be made on recognizing what are the points of bottleneck. When performing anomaly detection, the administrator can monitor for unexpected changes to the causal graph, indicating that the performance of certain metrics has changed significantly.

C. Problem Detection: Prediction

Finally, consider the problem of accurately forecasting resource utilization within the system. To address this difficulty, the final component is responsible for performing prediction on the time series metrics. The input to this component consists of the metrics to perform the regression on, and a dependency graph as computed by the previous component. Because the causal metrics present a more rich feature set, the addition of the dependency graph aids in the accuracy of the regression. The KPI data to be predicted is sent through a preprocessing step which constructs a feature space composed of all causally related metrics. The output of this component is a ML model which is capable of predicting future metrics based on a sequential input of time series data from the corresponding nodes.

Using the causal metrics combined with deep learning techniques allows for *Minerva* to capture previously unrecognizable non-linear relationships between this multivariate input space.

Design Requirement: For the prediction component, the algorithm must be able to do (1) create a model based on historic data in order to make accurate estimates about the future, (2) incorporate dependency information to improve the accuracy of forecasts (multivariate analysis), and (3) update the model online so as to be able to adapt to changes in the system.

Management Use Case: Predictions computed by this component can be used for *resource planning* and *anomaly detection*. Because this prediction can be run on any node or metric within the system, the predictive capabilities of the model can be useful for determining future demands, and to take appropriate actions. For instance, if the model recognizes future spikes in incoming traffic, the system can implement an auto-scaling policy to account for the future load. Anomaly detection can be implemented by checking the actual observed output against the predicted output. Significant divergences which fall outside of the expected accuracy and standard deviation can be considered anomalous behavior.

V. System Framework

We have designed and deployed a prototype implementation of *Minerva*, which runs as part of a container-based service, monitoring and diagnosing problems in the deployed system containers. *Minerva* collects metrics from the containers using a local statsd agent, running on each node, and stores the metrics in a time-series database e.g. opentsdb [5]. *Minerva* communicates with the database through Grafana [21], a query service which allows for the KPI data to be pulled and visualized. After performing the analysis, *Minerva* stores the outputs and presents them, in a visual manner, to the system administrator, thus allowing them to better perform different network management tasks. The overall implementation of this system can be observed in Figure 1. As stated previously, this design allows for *Minerva* to be minimally invasive and readily deployable in any existing distributed system/network.

Next, we briefly present and motivate the specific algorithms used to realize *Minerva*'s learning pipeline.

Dimensionality Reduction: Clustering Our premise is that if some metrics strongly correlate with each other, then it might not be necessary to consider all of them when making decisions about the control of the application. For this, we use the k-Shape clustering algorithm [32].

k-Shape is a recent clustering algorithm that is applied to time-series even if one lags the other, and can scale linearly with the number of metrics. The scale and shift invariance that this approach provides is key to our application. For example, CPU (measured in %) and memory (measured in bytes) may be highly correlated, but their amplitudes will certainly need normalization. Any lag is correlation is also adjusted for in this technique. Other approaches such as Principal Component Analysis (PCA) and random projections can also be used for dimensionality reduction. However we chose clustering approach since it provides us with better interpretability, scalability and stability across multiple runs compared to other techniques. Clustering results can be visually inspected by developers, who can also use any application-level knowledge to validate their correctness. Additionally, the amount of change between two iterations of the clustering algorithm can be measured (e.g.: Jaccard similarity coefficient [31]) and significant changes to cluster compositions can be flagged.

The representative metrics produced by the clustering step are then used in conjunction with the causal analysis graph obtained in the next step to identify and understand the relationships across components.

Causality Extraction: We hypothesize that nodes which display causal dependencies should be analyzed together when performing root cause analysis, anomaly detection, and prediction. To detect these dependencies, we employ Granger causality [22] for pairwise comparisons which is useful in determining whether a time series can be useful in predicting another time series. It offers a statistical approach in understanding relationships across components of a microservices application, in which the interaction of the components closely follows the request path of the application and can therefore be predictive of the changes in the metrics of the components in the path. The appearance/disappearance of relationships between clusters and/or changing attributes (e.g., Granger causality time lag) can be an indication of an anomaly, because one would expect cluster similarity to imply the maintenance of cluster relationships and causal relationships.

While Granger causality requires pairwise causality between all-metrics, a traditionally computationally expensive task, our dimensionality reduction ensures that the data is reduced to ensure that pairwise computations are tractable. Further, we create distributed implementations using Apache Spark to speed up clustering and causality algorithms.

Prediction: Our final hypothesis is that forecasting service metrics is extremely challenging in a large system with a microservices-based architecture due to the complex dependencies between the different system services, and non-linear dependencies spanning across different time lags. Our prior systems with univariate statistical models were woefully error-prone, and not amenable to rapid changes that are the norm in cloud deployments. To this extent, we perform the prediction using deep learning techniques which is able to better interpret these unconventional relationships and produce stronger results than classical regression models.

We employ a deep learning model which consists of a Convolutional Neural Network (CNN) [24] in sequence with a Recurrent Neural Network (RNN) [39]. We transform the system graph into a spatial tensor, where each node corresponds to a cell within a matrix, and fill these cells with the node's time series metrics. We compute the embeddings using the the node2vec [23] algorithm which preserves the spatial relationships between nodes which display homophily and structural equivalence. This transformation allows for the model to interpret both the spatial and temporal information from the system to improve its regression accuracy and performance.

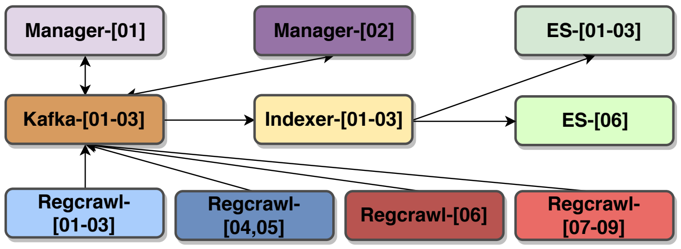

Figure 2: This figure shows the architecture and dependencies of our Vulnerability Analysis cloud service. The nodes are aggregated together based on service similarity. The accuracy of the clusters is confirmed by an individual with expert knowledge of the system.

VI. Evaluation

We evaluate our framework by running *Minerva* on metrics collected from a system running a collection of services. In our evaluation, we seek to answer the question *how effectively can unsupervised techniques be used to make inferences about a distributed system?* We present results detailing the efficacy of each framework component at extrapolating information about the system, and present a use-case showing the ability of *Minerva* to detect service-level failures.

Evaluation Setup: The system architecture can be observed in Figure 2. The system is composed of 21 nodes running 24 different services and is deployed on IBM Bluemix [2] platform. Each service runs within a container, while each node is responsible for running one or more services.

For our analysis, we collect the following key performance indicators (KPIs) at a service-level granularity over the duration of one week: cpu-user, cpu-system, cpu-idle, and memory-utilization. Each node runs a statsd server allowing for the metrics to be periodically pulled and aggregated together. While our current implementation performs on offline analysis, we envision *Minerva* running in an online environment, where it is capable of reacting to the dynamically changing components of the system.

We initially implemented the whole pipeline in python for rapid development and testing, but also created a fully distributed implementation in scala on the Apache Spark platform to scale for production use.

1. Clustering: As a first step, we filter out metrics that have low variance indicating they are not changing much over time. We use the k-Shape algorithm described in Section V to cluster the KPI metrics. While the results are omitted for brevity, we use the Adjusted Mutual Information score to understand how *consistent* the clustering across different runs is, and we try to validate whether the metrics in a cluster indeed belong together.

Using this methodology, the clustering component successfully reconstructs 9 out of the 10 clusters listed in

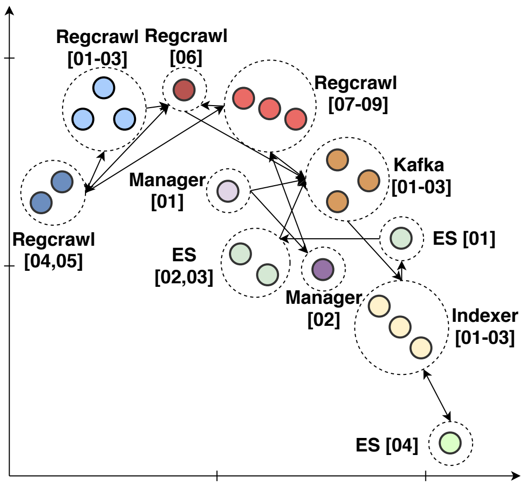

Figure 3: This figure shows the result of the spatial node embedding, clustering, and causal analysis.

Figure 2. The methodology fails to group ES-01 with the cluster of ES-02 and ES-03, instead, isolating the node from all other clusters. Further analysis reveals that ES-01 displays a highly periodic cpu-usage pattern which differs significantly from that of ES-02 and ES-03. Although they share similar memory-utilization patterns, the differing cpu-usage behavior validates the necessity of keeping these nodes separate. The complete results of the clustering can be observed in Figure 3.

2. Causal Analysis: We perform causal analysis using the service clusters computed in the previous stage, and extrapolate the discovered relationships to all services within a given cluster. The output of this step is a service dependency graph based on observable metrics.

We compute the Granger causality [22] between the centroids for the metrics of each cluster. We add an edge between two nodes if the granger test reveals a high probability of causality (p ¡ 0.01). Once the test has finished, we compute the overall edge weight between two nodes as the total number of discovered edges divided by the total number of possible edges. As can be observed in Figure 3, the computed inference graph preserves the overall structure and dependencies of the system graph down in Figure 2. In the inference graph, the causal edges indicate a control flow from Regcrawl to Kafka, Kafka to Indexer, and Indexer to ES.

While we chose the least intrusive method to infer the causal graph in this system, we can use the causal graph obtained through any other method (e.g.: tracing, instrumentation) in the next step.

3. Forecasting: We use the forecasting component to perform KPI prediction on a per-service basis. We use an ensemble learning method comprising a Deep Neural Network (DNN) architecture to perform regression and an online ARIMA [28] model (state-of-the-art statistical forecaster). The DNN forecaster is a Convolutional Neural Network [24] to create spatial embeddings [23] on the causal graph, followed by a regression-based Recurrent Neural Network [39] to provide time-series based forecasting on the metrics data. To improve the performance of the deep learning models, we structure the input space to consist of the causal metrics discerned in the previous step, and use the inference graph to compute the discrete node embeddings. This forecaster runs alongside an ARIMA forecaster which is adept at adjusting to short term fluctuations in the presence of cloud dynamics. The ensemble approach is composed of a weighted average between the two forecast models, where the weights are computed based on the MSE of each forecaster. We plan to create a longer technical report with details on these techniques.

Figure 4 shows the results of running the regression on the following services: Indexer-[01-03]. When the system is operating ”normally”, the forecasting component is able to make predictions with a mean squared error of 0.107, 0.153, and 0.120 respectively for each service. Compared to the output of the ARIMA model which has MSEs of 0.233, 0.285, and 0.233, it can be asserted that the deep learning and spatial embedding approach produces significantly stronger results.

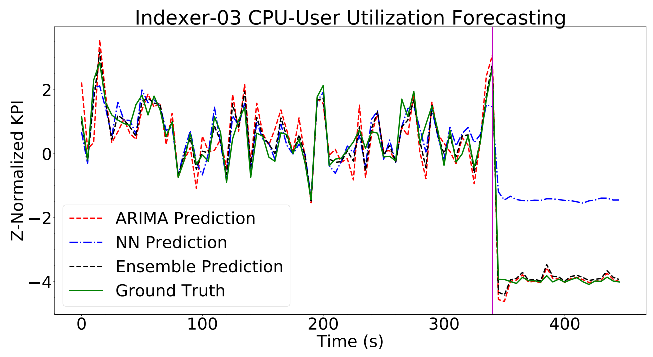

Figure 4: The figure shows the results of the regression analysis. The left region displays the ability of *Minerva* to make predictions while the system operates normally. The right region displays the ability for *Minerva* to detect anomalous behavior.

4. Service-Level Failure: We evaluate the ability of *Minerva* to identify anomalous behavior by introducing a service-level failure within the system. We produce this behavior by shutting down the primary service running on the Indexer nodes, simulating a multi-point failure. Using *Minerva*, we monitor the expected behavior of each node by comparing the output of the prediction against the observed KPI metrics. We expect any significant divergence between the predicted output and the observed output to indicate anomalous behavior.

Figure 4 shows the output of the regression for the Indexer-03 node when the fault occurs. As can be observed, the prediction for each node significantly deviates from the observed value around the 320 second mark. Monitoring the MSE between the observed and predicted values shows a significant divergence once the fault occurs.

Model	MSE-Before	MSE-After
Arima	0.35	0.26
CNN+RNN	0.16	6.31
Ensemble	0.16	0.24

Table I: Mean squared error (MSE) before and after the injected anomaly for each of the three models.

As can be observed in Table I, the deep learning approach produces stronger results during “normal” system operation, while also being able to indicate significant divergence from expected behavior which is key for performing anomaly detection. Once the anomaly is detected, the ensemble approach starts to heavily favor the online ARIMA model because of its ability to quickly adapt to the new data regime.

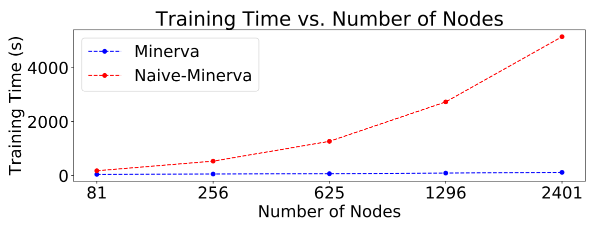

Figure 5: DNN training times for Minerva (subset of dataset) and Naive-Minerva (full dataset).

5. Benefit of Dimensionality Reduction: Figure 5 gives a comparison of running Minerva’s forecasting component with dimensionality reduction against a “naive” forecasting without dimensionality reduction or causality. We observe that Minerva is able to reduce training time by 91%: this performance speed up occurs because Minerva is able to focus on a subset of the data for training by identifying strongly correlated and causal subsets of nodes.

VII. Conclusion

In this paper, we have presented a framework for managing cloud based systems by leveraging unsupervised techniques to parse through large metric datasets. We have defined the different components of the framework, how they should communicate, and how the output of each component can be used to perform specific system management tasks. We show that *Minerva* is able to make strong inferences about correlated and causally connected nodes which reveals previously unknown information about the system.

VIII. Acknowledgements

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and

distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] Elastic search. https://www.elastic.co.

[2] IBM Bluemix. https://www.ibm.com/cloud-computing/bluemix/.

[3] Kibana. https://www.elastic.co/products/kibana.

[4] Logstash. https://www.elastic.co/products/logstash.

[5] opentsdb. http://opentsdb.net/.

[6] Atlas. https://github.com/Netflix/atlas/wiki/Overview, 2015.

[7] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[8] Amazon Web Services. Autoscaling. https://aws.amazon.com/autoscaling/.

[9] Amazon Web Services. Cloudwatch. https://aws.amazon.com/cloudwatch/.

[10] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 13–24. ACM, 2007.

[11] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[12] C. Canali and R. Lancellotti. An adaptive technique to model virtual machine behavior for scalable cloud monitoring. In *Computers and Communication (ISCC), 2014 IEEE Symposium on*, pages 1–7. IEEE, 2014.

[13] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 595–604. IEEE, 2002.

[14] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *OSDI*, pages 217–231, 2014.

[15] G. da Cunha Rodrigues, R. N. Calheiros, M. B. De Carvalho, C. R. P. dos Santos, L. Z. Granville, L. Tarouco, and R. Buyya. The interplay between timeliness and scalability in cloud monitoring systems. In *Computers and Communication (ISCC), 2015 IEEE Symposium on*, pages 776–781. IEEE, 2015.

[16] Etsy. statsd. https://github.com/etsy/statsd, s.

[17] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[18] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. Ieee, 2010.

[19] Google. Deepmind. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.

[20] Google Cloud Platform. Stack driver. https://cloud.google.com/stackdriver/.

[21] Grafana Labs. Grafana. https://grafana.com/.

[22] C. W. Granger. Causality, cointegration, and control. *Journal of Economic Dynamics and Control*, 12(2-3):551–559, 1988.

[23] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[25] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *HotNets*, pages 50–56, 2016.

[26] S. Meng and L. Liu. Enhanced monitoring-as-a-service for effective cloud management. *IEEE Transactions on Computers*, 62(9):1705–1720, 2013.

[27] Microsoft. Monitor. https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview.

[28] R. F. Nau. Arima. http://people.duke.edu/~rnau/411arim.html.

[29] Netflix. Vector. https://github.com/Netflix/vector, 2016.

[30] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, volume 13, pages 69–82, 2013.

[31] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1, 2013.

[32] J. Paparrizos and L. Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1855–1870. ACM, 2015.

[33] H.-S. Park and C.-H. Jun. A simple and fast algorithm for k-medoids clustering. *Expert systems with applications*, 36(2):3336–3341, 2009.

[34] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[35] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[36] Uber. Observability at uber engineering: Past, present, future. https://www.youtube.com/watch?v=2JAnmzVwgP8.

[37] Uber. Argos. https://eng.uber.com/argos/, 2015.

[38] K. Wagstaff, C. Cardie, S. Rogers, S. Schrödl, et al. Constrained k-means clustering with background knowledge. In *ICML*, volume 1, pages 577–584, 2001.

[39] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[40] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653, June 2005.