Ward and Barker *Journal of Cloud Computing: Advances, Systems and Applications* (2014) 3:24
DOI 10.1186/s13677-014-0024-2

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Observing the clouds: a survey and taxonomy of cloud monitoring

Jonathan Stuart Ward[†] and Adam Barker[*†]

Abstract

Monitoring is an important aspect of designing and maintaining large-scale systems. Cloud computing presents a unique set of challenges to monitoring including: on-demand infrastructure, unprecedented scalability, rapid elasticity and performance uncertainty. There are a wide range of monitoring tools originating from cluster and high-performance computing, grid computing and enterprise computing, as well as a series of newer bespoke tools, which have been designed exclusively for cloud monitoring. These tools express a number of common elements and designs, which address the demands of cloud monitoring to various degrees. This paper performs an exhaustive survey of contemporary monitoring tools from which we derive a taxonomy, which examines how effectively existing tools and designs meet the challenges of cloud monitoring. We conclude by examining the socio-technical aspects of monitoring, and investigate the engineering challenges and practices behind implementing monitoring strategies for cloud computing.

Keywords: Cloud computing; Monitoring

Introduction

Monitoring large-scale distributed systems is challenging and plays a crucial role in virtually every aspect of a software orientated organisation. It requires substantial engineering effort to identify pertinent information and to obtain, store and process that information in order for it to become useful. Monitoring is intertwined with system design, debugging, troubleshooting, maintenance, billing, cost forecasting, intrusion detection, compliance, testing and more. Effective monitoring helps eliminate performance bottlenecks, security flaws and is instrumental in helping engineers make informed decisions about how to improve current systems and how to build new systems.

Monitoring cloud resources is an important area of research as cloud computing has become the de-facto means of deploying internet scale systems and much of the internet is tethered to cloud providers [1,2]. The advancement of cloud computing, and by association cloud monitoring, is therefore intrinsic to the development of the next generation of internet.

Cloud computing has a unique set of properties, which adds further challenges to the monitoring process. The most accepted description of the general properties of cloud computing comes from the US based National Institution of Standards and Technology (NIST) and other contributors [3,4]:

- **On-demand self service:** A consumer is able to provision resources as needed without the need for human interaction.
- **Broad access:** Capabilities of a Cloud are accessed through standardised mechanisms and protocols.
- **Resource Pooling:** The Cloud provider's resources are pooled into a shared resource which is allocated to consumers on demand.
- **Rapid elasticity:** Resources can be quickly provisioned and released to allow consumers to scale out and in as required.
- **Measured service:** Cloud systems automatically measure a consumers use of resources allowing usage to be monitored, controlled and reported.

Despite the importance of monitoring, the design of monitoring tools for cloud computing is as yet an under researched area, there is hitherto no universally accepted toolchain for the purpose. Most real world cloud monitoring deployments are a patchwork of various data collection, analysis, reporting, automation and decision making

*Correspondence: adam.barker@st-andrews.ac.uk
[†]Equal contributors
School of Computer Science, University of St Andrews, St Andrews, UK

software. Tools from HPC, grid and cluster computing are commonly used due to their tendencies towards scalability while enterprise monitoring tools are frequently used due to their wide ranging support for different tools and software. Additionally, cloud specific tools have begun to emerge which are designed exclusively to tolerate and exploit cloud properties. A subclass of these cloud monitoring tools are monitoring as a service tools which outsource much of the monitoring process a third party.

While the challenges associated with cloud monitoring are well understood, the designs and patterns which attempt to overcome the challenges are not well examined. Many current tools express common design choices, which affect their appropriateness to cloud monitoring. Similarly there are a number of tools which exhibit relatively uncommon designs which are often more appropriate for cloud computing. Arguably, due to the compartmentalisation of knowledge regarding the design and implementation of current tools, emerging tools continue to exhibit previously employed schemes and demonstrate performance similar to well established tools. We therefore contend that it is necessary to examine the designs common to existing tools in order to facilitate discussion and debate regarding cloud monitoring, empower operations staff to make more informed tool choices and encourage researchers and developers to avoid reimplementing well established designs.

In order to perform this investigation we first present a set of requirements for cloud monitoring frameworks, which have been derived from the NIST standard and contemporary literature. These requirements provide the context for our survey in order to demonstrate which tools meet the core properties necessary to monitor cloud computing environments. Pursuant to this, we perform a comprehensive survey of existing tools including those from multiple related domains. From this comprehensive survey we extract a taxonomy of current monitoring tools, which categories the salient design and implementation decisions that are available. Through enumerating the current monitoring architectures we hope to provide a foundation for the development of future monitoring tools, specifically built to meet the requirements of cloud computing.

The rest of this paper is structured as follows: Section 'Monitoring' provides an overview of the traditional monitoring process and how this process is applied to cloud monitoring. Section 'Motivation for cloud monitoring' describes the motivation for cloud monitoring and details the behaviours and properties unique to cloud monitoring that distinguish it from other areas of monitoring. Section 'Cloud monitoring requirements' presents a set of requirements for cloud monitoring tools derived from literature, which are used to judge the effectiveness of current monitoring tools and their appropriateness to cloud monitoring. Section 'Survey of general monitoring systems' surveys monitoring systems which predate cloud computing but are frequently referred to as tools used in cloud monitoring. Section 'Cloud monitoring systems' surveys monitoring tools which were designed, from the outset for cloud computing. Section 'Monitoring as a service tools' surveys a monitoring as a service tools, a recent development which abstracts much of the complexity of monitoring away from the user. Section 'Taxonomy' extrapolates a taxonomy from the surveyed tools and categorises them accordingly. We then use this taxonomy to identity how well current tools address the issues of cloud monitoring and analyse the future potential of cloud monitoring tools. Section 'Monitoring as an engineering practice' considers monitoring from a practical standpoint, discussing the socio-technical and organisational concerns that underpin the implementation of any monitoring strategy. Finally Section 'Conclusion' concludes this paper.

Monitoring

At its very simplest monitoring is a three stage process illustrated by Figure 1: the collection of relevant state, the analysis of the aggregated state and decision making as a result of the analysis. The more trivial monitoring tools are simple programs which interrogate system state such as the UNIX tools *df*, *uptime* or *top*. These tools are run by a user who in turn analyses the system state and makes an informed decision as to what, if any action to take. Thus, in fact, the user is performing the vast majority of the monitoring process and not software. As computing systems continue to grow in size and complexity there is an increasing need for automated tools to perform monitoring with a reduced, or removed need for human interaction. These systems implement all or some of the 3 stage monitoring process. Each of these stages have their own challenges, especially with regards to cloud computing.

Collection

All monitoring systems must perform some form of data collection. In tools commonly used for monitoring commodity server deployments this is achieved through the use of a monitoring server. In these common use cases a monitoring server either actively polls state on remote hosts, or remote hosts push their state to the server. This mechanism is ideal for small server deployments: it is simple, it has no indirection and it is fast. As the number of monitored servers grow, data collection becomes increasingly challenging. The resources of a single machine eventually become insufficient to collect all the required state. Additionally, in tightly coupled systems where there is often an active human administrator, the

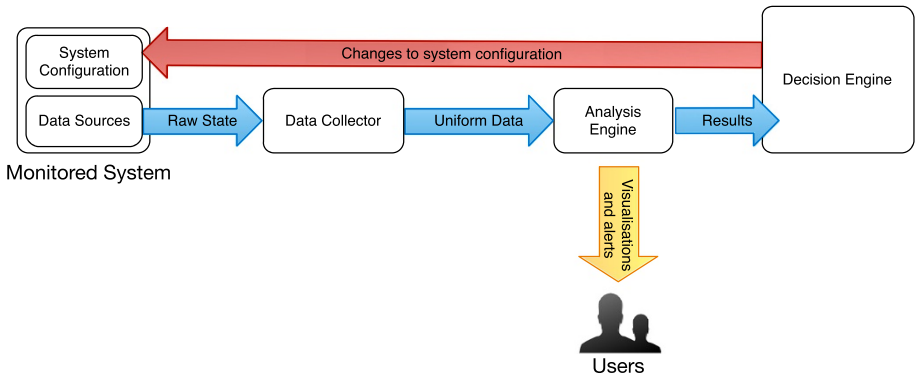

Figure 1 The three stage monitoring process.

associated configuration overhead becomes prohibitive; requiring frequent interaction. These challenges have led to the development of a number of schemes to improve the scalability of monitoring systems.

There are a diverse set of methods for collecting monitoring state from cloud deployments. Many tools still rely upon fully centralised data collection, while others have extended this design through the use of trees and other forms of overlay.

Data collection trees are the simplest means of improving scalability over fully centralised systems. Monitoring architectures using trees to collect and propagate data have improved scalability when compared to fully centralised but still rely upon single points of failure. Typically, a central monitoring server sits at the root of the tree and is supported by levels of secondary servers which propagate state from monitored hosts (the leaves of the tree) up to the root. Failure of a monitoring server will disrupt data collection from its subtree.

Trees are not the solution to all scalability issues. In the case of large scale systems, most tree schemes require a significant number of monitoring servers and levels of the tree in order to collect monitoring information. This requires the provisioning of significant dedicated monitoring resources, which increases the propagation latency; potentially resulting in stale monitoring data at the top of the hierarchy.

With large scale systems becoming more common place, several recent monitoring systems have abandoned centralised communication models [5-8]. A new and diverse class of monitoring system makes use of peer to peer concepts to perform fully decentralised data collection. These systems make use of distributed hash tables, epidemic style communication and various P2P overlays to discover and collect data from machines. Decentralised schemes have inherent scalability improvements over earlier schemes but this is not without a series of additional challenges including: the bootstrap problem, lookup, replication and fault tolerance. Decentralised systems must overcome these challenges and risk becoming slower and more cumbersome than their centralised counterparts.

A very recent development is monitoring as a service: SaaS applications that abstract much of the complexity of monitoring away from the user. This class of monitoring tool, presumably, makes use of similar architectures to existing tools but introduces a novel separation of concerns between where data is generated and where it is collected. In these systems a users installs a small agent which periodically pushes monitoring state to a service endpoint, all functionality beyond that is the prerogative of the monitoring provider. This alleviates complexity on behalf of the user but exacerbates the complexity of the service provider who must provide multi-tenanted monitoring services.

Analysis

Once data has been collected it must be analysed or otherwise processed in order for it to become useful. The range of analysis offered by monitoring tools varies greatly from simple graphing to extremely complex system wide analysis. With greater analysis comes greater ability to detect and react to anomalous behaviour, this however comes with an increased computational cost.

Graphing is the lowest common denominator of analysis and offloads the complexity of analysis to the end user. Resource monitors and systems, which exclusively collect time series data are the only tools which tend to rely solely on graphing. These tools typically provide a holistic view of system wide resource usage; allowing a user to interrogate machine specific resource usage if necessary. It is then up to the user to detect resource spikes, failures and other anomalous behaviour.

Other monitoring systems provide more complex analysis. Threshold analysis is the most common form of analysis, found in the vast majority of monitoring systems.

Threshold analysis is where monitoring values are continually checked against a predefined condition, if the value violates the condition an alert is raised or other action taken. This basic strategy is used to provide health monitoring, failure detection and other forms of basic analysis.

Threshold monitoring allows for expected error states to be detected but does not provide a means to detect unexpected behaviour. As cloud deployments become increasingly large and complex the requirement for automated analysis becomes more pressing. To this end various recent tools provide facilities for trend analysis and stream processing to detect anomalous system states and complex error conditions beyond what simple threshold analysis is capable of detecting. These more complex analysis tasks often require bespoke code, which runs outside of the monitoring environment consuming data via an API or alternatively runs as part of the monitoring system through a plugin mechanism.

Complex analysis is typically more resource intensive than simple threshold, analysis requiring significant memory and CPU to analyse large volumes of historical data. This is a challenge for most monitoring systems. In centralised monitoring system a common mitigation is simply provisioning additional resources as necessary to perform analysis. Alternatively, taking cues from volunteer computing, various monitoring systems attempt to schedule analysis over under utilised hosts that are undergoing monitoring.

Decision making

Decision making is the final stage of monitoring and is particularly uncommon in the current generation of monitoring tools. As previously discussed, simple tools collect and graph analysis requiring the user to analyse the current state of the system and in turn make any appropriate decisions. This is a challenge for the user as it requires them to consider all known state, identify issues and then devise a set of actions too rectify any issues. For this reason, all major organisations employ significant operations personnel in order to enact any appropriate actions. Current monitoring tools are intended to support monitoring personnel rather than to take any action directly.

Many current monitoring tools support the notion of event handlers; custom code that is executed dependant upon the outcome of analysis. Event handlers allow operations personnel to implement various automated strategies to prevent errors cascading, their severity increasing or even resolve them. This represents an automation of part of the manual error handling process and not a true autonomic error correction process.

Some monitoring tools provide mechanisms to implement basic automated error recover strategies [9,10]. In the case of cloud computing the simplest error recovery mechanism is to terminate a faulty VM and then instantiate a replacement. This will resolve any errors contained to a VM (stack overflows, kernel panics etc) but will do little to resolve an error trigged by external phenomenon or an error that continuously reoccurs.

A true autonomic monitoring system which can detect unexpected erroneous states and then return the system to an acceptable state remains an open research area. This area of research is beyond the scope of most other monitoring challenges and exists within the domain of self healing autonomic systems.

Motivation for cloud monitoring

Monitoring is an important aspect of systems engineering which allows for the maintenance and evaluation of deployed systems. There are a common set of motivations for monitoring which apply to virtually all areas of computing, including cloud computing. These include: capacity planning, failure or under-performance detection, redundancy detection, system evaluation, and policy violation detection. Monitoring systems are commonly used to detect these phenomena and either allow administrators to take action or to take some form of autonomous action to rectify the issue. In the case of cloud computing there are additional motivations for monitoring which are more unique to the domain, these include:

Performance uncertainty

At the infrastructure layer, performance can be incredibly inconsistent [11] due to the effects of multi-tenancy. While most IaaS instance types provide some form of performance guarantee these typically come in the nebulous form of a 'compute unit'. In the case of the Amazon Compute Unit this is defined as: "the relative measure of the integer processing power of an Amazon EC2 instance" [12] and in the case of the Google Compute Unit as: "a unit of CPU capacity that we use to describe the compute power of our instance types. We chose 2.75 GCEUs to represent the minimum power of one logical core (a hardware hyper-thread) on our Sandy Bridge platform" [13]. These measurements give little in the way of absolute performance metrics and at best serve to give a vague indication of performance levels. Worse still are the smallest instance types: t1.micro in the case of Amazon and f1-micro in the case of Google. These instance types have no stated performance value in terms of compute units, or indeed otherwise, and are particularly susceptible to the effects of cpu stealing. CPU stealing [14] is an emergent property of virtualization which occurs when the hypervisor context switches a VM off the CPU. This occurs based on some sort of policy: round robining, demand based, fairness based or other. From the perspective of the VM, there is a period where no computation, or indeed any activity, can occur. This prevents any real time

applications from running correctly and limits the performance of regular applications. The performance impact of CPU stealing can be so significant that Netflix [15] and other major cloud users implement a policy of terminating VMs that exceed a CPU stealing threshold as part of their monitoring regime.

The vague notion of compute units gives little ability to predict actual performance. This is primarily due to two factors: multi-tenancy and the underlying hardware. Virtualization does not guarantee perfect isolation, as a result users' can affect one another. Many cloud providers utilise (or are believed to utilise) an over-subscription model whereby resources are over sold to end users. In this case, users will effectively compete for the same resources resulting in frequent context switching and overall lower performance. The exact nature of this mechanism and the associated performance implications are not disclosed by any cloud provider. This issue is further compounded by the underlying hardware. No cloud provider has homogeneous hardware. The scale that cloud providers operate at makes this impossible. Each major cloud provider operates a wide range of servers with different CPUs. The underlying hardware has the most crucial effect upon performance. Again, no cloud provider discloses information about their underlying hardware and the user has no knowledge of the CPU type that their VM will have prior to instantiation. Thus, details critical to determining the performance of a VM are unavailable until that VM has been instantiated.

Figure 2 gives some example as to the range of performance variation commonly found in cloud deployments. This Figure shows the results of an Apache benchmark performed on 5 EC2 VMs instantiated at the same time, in the same US East Region. Each of these VMs are of the m1.medium type. In this sample of 5 instances, three CPU types were found: the Intel Xeon E5645, E5430 and E5507. An Apache benchmarking test was performed 12 separate times per instance over a 3 hour period in order to ascertain how many http requests per second (a predominantly CPU bound activity) each instance could fulfil. As is evident from the graph, a range of performance was demonstrated. The lowest performance was exhibited by instance C which in one test achieved 2163 requests per second. The highest performance was demonstrated by instance E which achieved 3052 requests per second. This represents a 29% difference between the highest and lowest performing VMs. Interesting to note is the processor responsible for delivering the best and second best performance, the E5430, is an older end of line chip. The newer CPU models, which were expected to yield greater performance handled demonstrably fewer tests. Whether this is due to the underlying physical machines being oversold or due to some other multi-tenanted phenomena is unclear. Notable is instance C which yielded a significant range of performance variation, again for an unclear reason. This trivial benchmarking exercise is comparable with results found in literature [11,11,16] and demonstrates the range of performance that can be found from 5 VMs that are expected to be identical. In any use case involving a moderate to high traffic web application several of these instances may prove unsuitable. Without the availability of monitoring data it is impossible for stakeholders to identify these issues and thus will suffer from degraded performance.

The non-availability of exact performance metrics makes deploying certain use cases on the cloud a challenge. An example of this is HPC applications. Cloud computing has frequently been touted as a contender for supporting the next generation of HPC applications. Experiments showing that it is feasible to deploy a comparable low cost supercomputer capable of entering the top500 [17] and the recent availability of HPC instance types makes cloud computing an appealing choice for certain HPC applications. Performance uncertainty, however, makes it hard to create a deployment with the desired level of performance. Conceptually the same size

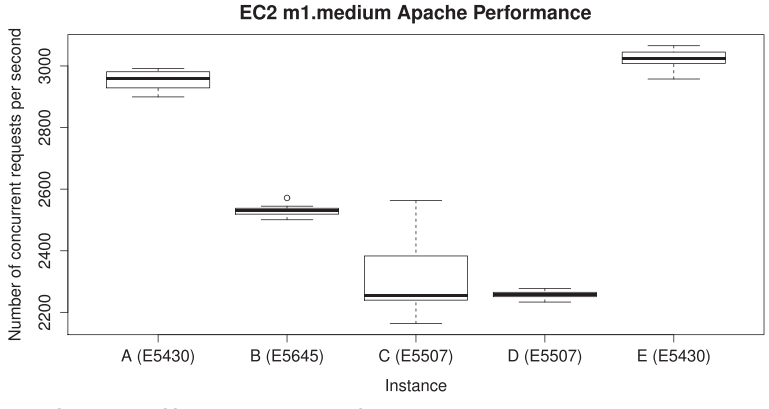

Figure 2 Apache requests served per second by AWS EC2 m1.medium instances.

of deployment could offer significantly different rates of performance at different points in time which is clearly undesirable for compute bound applications. These concerns will appear in any cloud deployment which expects a certain level of performance and may be prohibitive.

This uncertainty makes cloud monitoring essential. Without monitoring it is impossible to understand and account for the phenomena mentioned above. To summarise, the need for cloud monitoring, with regards to performance certainty is four fold:

- To quantify the performance of a newly instantiated VM deployment to produce an initial benchmark that can determine if the deployment offers acceptable performance.
- In order to examine performance jitter to determine if a deployment is dropping below an acceptable baseline of performance.
- To detect stolen CPU, resource over-sharing and other undesirable phenomena.
- To improve instance type selection to ensure that the user achieves best performance and value.

SLA enforcement

With such a dependency upon cloud providers, customers rely upon SLAs to ensure that the expected level of services are delivered. While downtime or non availability is easily detected there are other forms of SLA violation which are not easily noticed. High error rates in APIs and other services and performance degradation of VMs and services are not always easily detectable but can have significant impact upon an end users deployments and services. Monitoring is therefore essential in order to guarantee SLA compliance to produced the necessary audit trail in the case of SLA violation. Monitoring is also important on the part of the cloud provider, in this capacity, to ensure SLA compliance is maintained and that an acceptable user experience is provided to customers.

In the case of recent outages and other incidents [18,19] cloud provider's SLAs have been ineffective in safeguarding performance or otherwise protecting users. Monitoring, therefore, becomes doubly important as it allows cloud users to migrate their architecture to an alternative provider or otherwise compensate when a cloud provider does not adhere to the level of expected service.

Defeating abstraction

Cloud computing is based on a stack which builds functionality on increasing levels of abstraction. It therefore seems counter-intuitive to attempt to circumvent abstraction and delve down into the level below. Doing so however allows users to become aware of phenomena that, whether they are aware of it or not, will affect their applications. There are a number of phenomena which are abstracted away from the user that crucially affect their applications, these include:

Load balancing latency

Many cloud providers including Amazon Web Services, Google Compute Engine and Microsoft Azure include a load balancer which can distribute load between VMs and create additional VMs as necessary. Such load balancers are based upon undisclosed algorithms and their exact operation is unknown. In many cases the load balancer is the point of entry to an application and as such, success of an application rests, in part, on effective load balancing. A load balancer is ineffective when it fails to match the traffic patterns and instantiate additional VMs accordingly. This results in increased load on the existing VMs and increased application latency. Monitoring load balancing is therefore essential to ensure that the process is occurring correctly and that additional VMs are created and traffic distributed as is required. Successfully detecting improper load balancing allows the user to alter the necessary policies or utilise an alternative load balancer.

Service faults

Public clouds are substantial deployments of hardware and software spread between numerous sites and utilised by a significant user-base. The scale of public clouds ensures that at any given time a number of hardware faults have occurred. Cloud providers perform a noble job in ensuring the continuing operation of public clouds and strive to notify end users of any service disruptions but do not always succeed. An Amazon Web Services outage in 2012 disrupted a significant portion of the web [19]. Major sites including Netflix, Reddit, Pintrest, GitHub, Imgur, Forsquare, Coursera, Airbnb, Heroku and Minecraft were all taken off-line or significantly disrupted due to failures in various AWS services. These failures were initially detected by a small number of independent customers and weren't fully publicised until it became a significant issue. The few users who monitored these emergence of these faults ahead of the public announcement stood far greater chance of avoiding disruption before it was too late. Monitoring service availability and correctness is desirable when one's own infrastructure is entirely dependant upon that of the cloud provider.

Location

Public cloud providers typically give limited information as to the physical and logical location of a VM. The cloud provider gives a region or zone as the location of the VM which provides little more than a continent as a location. This is usually deemed as beneficial; users do not need to concern themselves with data centres or other low level abstractions. Hoover for applications which are latency sensitive or otherwise benefit from collocation the

unavailability of any precise physical or logical locations is detrimental. If a user wishes to deploy an application as close to users or a data source as possible and has a choice of numerous cloud regions it is difficult to make an informed decision with the limited information that is made available by cloud providers [20,21]. In order to make an informed decision as to the placement of VMs additional information is required. To make an informed placement the user must monitor all relevant latencies between the user or data source and potential cloud providers.

Cloud monitoring requirements

Cloud computing owes many of its features to previous paradigms of computing. Many of the properties inherent to cloud computing originate from cluster, grid, service orientated, enterprise, peer to peer and distributed computing. However, as an intersection of these areas, cloud computing is unique. The differences between cloud monitoring and previous monitoring challenges has been explored in literature [22,23]. A previous survey of cloud monitoring [24] has addressed the challenges inherent to cloud computing. Examination of the challenges presented in the literature in combination with the well established NIST definition allow us to derive a set of requirements for a cloud monitoring framework. These requirements provide a benchmark against which to judge current systems. An ideal system will focus on and fulfil each of these requirements; realistically monitoring tools will focus predominantly upon one of these requirements with lesser or no support for the others. Importantly, we differentiate our survey from [24] by focusing upon the design and internals of current tools and taxonomies these tools accordingly, as opposed to focusing upon the requirements and open issues of cloud monitoring.

Scalable

Cloud deployments have an inherent propensity for change. Frequent changes in membership necessitate loose coupling and tolerance membership churn, while change in scale necessitates robust architectures which can exploit elasticity. These requirements are best encapsulated under the term scalability. A scalable monitoring system is one which lacks components which act as a bottleneck, single points of failure and supports component auto detection, frequent membership change, distributed configuration and management, or other features that allow a system to adapt to elasticity.

Cloud aware

Cloud computing has a considerable variety of costs - both in terms of capital expenditure and in terms of performance. Data transfer between different VMs hosted in different regions can incur significant financial costs, especially when dealing with big data [25,26]. Monitoring data will eventually be sent outside of the cloud in order to be accessed by administrators. In systems hosted between multiple clouds there will be both inter and intra cloud communication. Each of these cases have different costs and latencies associated with them both in terms of latency and in terms of financial cost to the user. Latency and QoS presents a significant challenge for applications running on the cloud [27,28] including monitoring. When monitoring physical servers a host can be but a few hops away, cloud computing gives no such guarantees. This will adversely affect any monitoring system which uses topologies which rely upon the proximity of hosts. A location aware system can significantly outperform a system which is not location aware [29] and reduce the costs inherently associated with cloud computing. Hence a cloud monitoring system must be aware of the location of VMs and collect data in a manner which minimizes delay and the costs of moving data.

Fault tolerance

Failure is a significant issue in any distributed system, however it is especially noteworthy in cloud computing as all VMs are transient [30]. VMs can be terminated by a user or by software and give no indication as to their expected availability. Systems must Current monitoring is performed based upon the idea that servers should be permanently available. As such current monitoring systems will report a failure and await the return of the failed server. A cloud monitoring system must be aware of failure, and of VM termination and account for it appropriately. It must not treat VM termination as failure but it must always detect genuine failure. Crucially failure, even widespread failure, must not disrupt monitoring.

Autonomic

Having to configure a live VM, even a trivial configuration, is a significant overhead when dealing with large numbers of VMs. When deployments auto-scale and rapidly change, human administrators cannot be required to perform any form of manual intervention. An autonomic system is one which has the capacity for self management; to configure and optimise itself without the need for human interaction. There are many different levels of autonomic behaviour from simple configuration management tools to self optimising and self healing systems. At the very least, a cloud monitoring system must require no signifiant configuration or manipulation at runtime. Greater degrees of autonomic behaviour is however, incredibly desirable.

Multiple granularities

Monitoring tools can collect a vast sum of data, especially from large and constantly changing cloud deployments.

Practically no data collected from systems is unimportant but it may be unrelated to a given use case. Some monitoring state may be useful only when analysed en masse while other state is immediately useful by itself. These factors necessitate a monitoring system which considers the multiple levels of granularity present in any monitored system. This can range from proving a mechanism to alter the rate at which data is collected or to dynamically alter what data is collected. It can also extend to tools which collect data from less conventional sources and from multiple levels of the cloud stack. Accounting for granularity when dealing with users is also essential. Users should not be expected to manually parse and analyse large sets of metrics or graphs. Monitoring tools should produce visualizations or other reports to the user at a coarse granularity appropriate for humans while collecting and analysing data at a much finer level of granularity.

Comprehensiveness

Modern systems consist of numerous types of hardware, VMs, operating systems, applications and other software. In these extremely heterogeneous environments there are numerous APIs, protocols and other interfaces which provide potentially valuable monitoring state. A cloud monitoring tool must be comprehensive: it must support data collection from the vast array of platforms, software and other data sources that comprise heterogeneous systems. This can be achieved either through the use of third party data collection tools, plugins or simply through extensive in built support.

Time sensitivity

Monitoring state can be useful long after it has been collected. Capacity planning, post mortem analysis and a variety of modelling strongly depends upon the availability of historical monitoring state. The more common functions of monitoring, such as anomaly detection, depend upon up to date monitoring state that is as close to real time as possible. Monitoring latency: the time between an phenomena occurring and that phenomena being detected, arises due to a number of causes. The data collection interval, the time between state being collected is a significant factor in monitoring latency. In some systems the collection interval is fixed, this is common in loosely coupled systems where monitored hosts push state at their own schedule. In other schemes the monitoring interval can be adjusted this is common in pull based schemes or in push schemes where there is some form of feedback mechanism. If the interval is fixed and events occur within the collection interval those events will remain undetected until the next collection occurs. In systems where the collection time increases as the number of monitoring hosts increases this can result in significant latency between phenomena occurring and their detection.

Monitoring latency can also be affected by communication overheads. Systems which rely upon conventional network protocols which provide no time guarantees can suffer increased latency due to packet loss or congestion. Encoding formats, data stores and various of data representation can also increase the time between state leaving its point or origin and it being analysed if they act as performance bottlenecks.

A monitoring system is time sensitive if it provides some form of time guarantee or provides a mechanism for reducing monitoring latency. These guarantees are essential to ensure continuous effective monitoring at scale or in the event of frequent change.

Survey of general monitoring systems

This section contains two categories of tools: tools developed before cloud computing and contemporary tools which were not designed specifically for cloud monitoring but have related goals. In the case of the former, these tools retain relevance to cloud computing either by utilising concepts pertinent to cloud monitoring or by being commonly utilised in cloud monitoring.

Ganglia

Ganglia [31] is a resource monitoring tool primarily intended for HPC environments. Ganglia organises machines into clusters and grids. A cluster is a collection of monitored servers and a grid is the collection of all clusters. A Ganglia deployment operates three components: Gmond, Gmetad and the web frontend. Gmond, the Ganglia monitoring daemon is installed on each monitored machine and collects metrics from the local machine and receives metrics over the network from the local cluster. Gmetad, the Ganglia Meta daemon polls aggregated metrics from Gmond instances and other Gmetad instances. The web frontend obtains metrics from a gmond instance and presents them to users. This architecture is used to form a tree, with Gmond instances at the leaves and Gmond instances at subsequent layers. The root of the tree is the Gmond instance which supplies state to the web frontend. Gmetad makes use of RRDTool [32] to store time series data and XDR [33] to represent data on the wire.

Ganglia provides limited analysis functionality. A third party tool: the Ganglia-Nagios bridge[34] allows Nagios to perform analysis using data collected by Ganglia. This attempts to gain the analysis functionality of Nagios while preserving the scalable data collection model of Ganglia. This is not a perfect marriage as the resulting system incurs the limitations of both tools but is often proposed as a stopgap tool for cloud monitoring. Similarly Ganglia can export events to Riemann.

Ganglia is first and foremost a resource monitor and was designed to monitor HPC environments. As such it

is designed to obtain low level metrics including CPU, memory, disk and IO. It was not designed to monitor applications or services and nor was it designed for highly dynamic environments. Plugins and various extensions are available to provide additional features but the requirements of cloud computing are very different to Ganglia's design goals. Despite this, Ganglia still sees some degree of usage within cloud computing due to its scalability.

Astrolabe

Astrolabe [8] is a tool intended for monitoring large scale distributed systems which is heavily inspired by DNS. Astrolabe provides scalable data collection and attribute based lookup but has limited capacity for performing analysis. Astrolabe partitions groups of hosts into an overlapping hierarchy of zones in a manner similar to DNS. Astrolabe zones have a recursive definition: a zone is either a host or a set of non overlapping zones. Two zones are non-overlapping if they have no hosts in common. The smallest zones consist of single hosts which are grouped into increasingly large zones. The top level zone includes all other zones and hosts within those zones. Unlike DNS, Astrolabe zones are not bound to a specific name server, do not have fixed attributes and state update are propagated extremely quickly.

Every monitored host runs the Astrolabe agent which is responsible for the collection of local state and the aggregation of state from hosts within the same zone. Each host tracks the changes of a series of attributes and stores them as rows in a local SQL database. Aggregation in Astrolabe is handled through SQL SELECT queries which serve as a form of mobile code. A user or software agent issues an query to locate a resource or obtain state and the Astrolabe deployment rapidly propagates the query through a peer to peer overly, aggregates results and returns a complete result. Astrolabe continuously recomputes these queries and returns updated results to any relevant clients.

Astrolabe has no publicly available implementation and the original implementation evaluates the architecture through the use of simulation. As a result, there is no definitive means to evaluate Astrolabe's use for cloud monitoring. Irrespective of this Astrolabe is an influential monitoring system which, unlike many contemporary, monitoring tools employs novel aggregation and grouping mechanisms.

Nagios

Nagios [35] is the de facto standard open source monitoring tool for monitoring server deployments. Nagios in its simplest configuration is a two tier hierarchy; there exists a single monitoring server and a number of monitored servers. The monitoring server is provided with a configuration file detailing each server to be monitored and the services each operates. Nagios then generates a schedule and polls each server and checks each service in turn according to that schedule. If servers are added or removed the configuration must be updated and the schedule recomputed. Plugins must be installed on the monitored servers if the necessary information for a service check cannot be obtained by interacting with the available services. A Nagios service check consists of obtaining the relevant data from the monitored host and then checking that value against a expected value or range of values; raising an alert if an unexpected value is detect. This simple configuration does not scale well, as the single server becomes a significant bottleneck as the pool of monitored servers grows.

In a more scalable configuration, Nagios can be deployed in an n-tier hierarchy using an extension known as the Nagios Service Check Acceptor (NCSA). In this deployment, there remains a single monitoring server at the top of this hierarchy, but in this configuration it polls a second tier of monitoring servers which can in turn poll additional tiers of monitoring servers. This distributes the monitoring load over a number of monitoring servers and allows for scheduling and polling to be performed in small subsections rather then en mass. The NCSA plugin that facilitates this deployment allows the monitoring results of a Nagios server to be propagated to another Nagios server. This requires each Nagios server to have its own independent configuration and the failure of any one of the monitoring servers in the hierarchy will disrupt monitoring and require manual intervention. In this configuration system administrators typically rely on a third party configuration management tool such as Chef or Puppet to manage the configuration and operation of each independent Nagios server.

A final, alternative configuration known as the Distributed Nagios Executor (DNX) introduces the concept of worker nodes [36]. In this configuration, a master Nagios server dispatches the service checks from its own schedule to a series of worker nodes. The master maintains all configuration, workers only require the IP address of the master. Worker nodes can join and leave in an ad hoc manner without disrupting monitoring services. This is beneficial for cloud monitoring; allowing an elastic pool of workers to scale in proportion to the monitored servers. If, however, the master fails all monitoring will cease. Thus, for anything other than the most trivial deployments additional failover mechanisms are necessary.

Nagios is not a perfect fit for cloud monitoring. There is an extensive amount of manual configuration required, including the need to modify configuration when monitored VMs are instantiated and terminated. Performance is an additional issue, many Nagios service checks are resource intensive and a large number of service checks

can result in significant CPU and IO overhead. Internally, Nagios relies upon a series of pipes, buffers and queues which can become bottlenecks when monitoring large scale systems [37]. Many of these host checks are, by default, non parallelised and block until complete. This severely limits the number of service checks that can be performed. While most, if not all of these issues can be overcome through the use of plugins and third party patches this requires significant labour. Nagios was simply never designed or intended for monitoring large scale cloud systems and therefore requires extensive retrofitting to be suitable for the task [38]. The classification of Nagios is dependant upon its configuration. Due to its age and extensive plugin library Nagios has numerous configurations.

Collectd

Collectd [39] is an open source tool for collecting monitoring state which is highly extensible and supports all common applications, logs and output formats. It is used by many cloud providers as part of their own monitoring solutions, including Rightscale [40]. One of the appeals of collectd is its network architecture; unlike most tools it utilises IPv6 and multicast in addition to regular IPv4 unicast. Collectd uses a push model, monitoring state is pushed to a multicast group or single server using one of the aforementioned technologies. Data can be pushed to several storage backends, most commonly RRDTool [32] is used but MongoDB, Redis, MySQL and others are supported. This allows for a very loosely coupled monitoring architecture whereby monitoring servers, or groups or monitoring servers, need not be aware of clients in order to collect state from them. The sophisticated networking and loose coupling allow collectd to be deployed in numerous different topologies, from simple two tier architectures to complex multicast hierarchies. This flexibility makes collectd one of the more popular emerging tools for cloud monitoring. Collectd, as the name implies, it primarily concerned with the collection and transmission of monitoring state. Additional functions, including the storage of state are achieved through plugins. There is no functionality provided for analysing, visualising or otherwise consuming collected state. Collectd attempts to adhere to UNIX principles and eschews non collection related functionality, relying on third party programs to provide additional functionality if required. Tools which are frequently recommended for use with Collectd include:

- Logstash [41] for managing raw logfiles, performing text search analysis which is beyond the perview of collected
- StatsD [42] for aggregating monitoring state and sending it to an analysis service
- Bucky [43] for translating data between StatsD, Collectd and Graphtite's formats.
- Graphite [44] for providing visualization and graphing
- drraw [45] an alternative tool for visualization RRDtool data
- Riemann [9] for event processing
- Cabot [46] for alerting

Collectd, in conjunction with additional tools make for a comprehensive cloud monitoring stack. There is however no complete, ready to deploy distribution of a collectd based monitoring stack. Therefore, there is significant labour required to build, test and deploy the full stack. This is prohibitive to organisations that lack the resources to roll their own monitoring stack. Colletd based monitoring solutions are therefore available only to those organisations that can develop and maintain their stack. For other organisations, a more complete solution or monitoring as a service tool may be more appropriate.

Riemann

Riemann [9] is an event based distributed systems monitoring tool. Riemann does not focus on data collection, but rather on event submission and processing. Events are representations of arbitrary metrics which are generated by clients and encoded using Google Protocol Buffers [47] and additionally contains various metadata (hostname, service name, time, ttl, etc). On receiving an event Riemann processes it through a stream. Users can write stream functions in a Clojure based DSL to operate on streams. Stream functions can handle events, merge streams, split streams and perform various other operations. Through stream processing Riemann can check thresholds, detect anomalous behaviour, raise alerts and perform other common monitoring use cases. Designed to handle thousands of events per second, Riemann is intended to operate at scale.

Events can be generated and pushed to Riemann in one of two ways. Either applications can be extended to generate events or third party programs can monitor applications and push events to Riemann. Nagios, Ganglia and collectd all support forwarding events to Riemann. Riemann can also export data to numerous graphing tools. This allows Riemann to integrate into existing monitoring stacks or for new stacks to be built around it.

Riemann is relatively new software and has developing a user base. Due it its somewhat unique architecture, scalability and integration Riemann is a valuable cloud monitoring tool.

sFlow and host sFlow

sFlow [48] is a monitoring protocol designed for network monitoring. The goal of sFlow is to provide an interoperable monitoring standard that allows equipment

from different vendors to be monitored by the same software. Monitored infrastructure runs an sFlow agent which receives state from the local environment and builds and sends sFlow datagram to a collector. A collector is any software which is capable of receiving sSlow encoded data. The original sFlow standard is intended purely for monitoring network infrastructure. Host sFlow [49] extends the base standard to add support for monitoring applications, physical servers and VMs. sFlow is therefore one of the few protocols capable of obtaining state from the full gamut of data centre technologies.

sFlow is a widely implemented standard. In addition to sFlow agents being available for most pieces of network equipment, operating systems and applications there are a large number of collectors. Collectors vary in terms of functionality and scalability. Collectors range from basic command line tools and simple web applications to large complex monitoring and analysis systems. Other monitoring tools discussed, including Nagios and Ganglia, are also capable of acting as sFlow collectors.

Rather than being a monitoring system, sFlow is a protocol for encoding and transmitting monitoring data which makes it unique amongst the other tools surveyed here. Many sFlow collectors are special purpose tools intended for DDOS prevention [50], network troubleshooting [51] or intrusion detection. At present, there is no widely adopted general purpose sFlow based monitoring tool. sFlow is however, a potential protocol for future monitoring solutions.

Logstash

Logstash is a monitoring tool quite unlike the vast majority of those surveyed here. Logstash is concerned not with the collection of metrics but rather with the collection of logs and event data. Logstash is part of the ElasticSearch family, a set of tools intended for efficient distributed real time text analytics. Logstash is responsible for parsing and filtering log data while other parts of the ElasticSearch toolset, notably Kibana (a browser based analytics tool) is responsible for analysing and visualising the collected log data. Logstash supports and event processing pipeline not dissimilar to Riemann allowing chains of filter and routing functions to be applied to events as they progress through the Logstash index.

Similar to other monitoring tools, Logstash runs a small agent on each monitored host referred to as a shipper. The shipper is a Logstash instance which is configured to take inputs from various sources (stdin, stderr, log files etc) and then 'ship' them using AMQ to an indexer. An indexer is another Logstash index which is configured to parse, filter and route logs and events that come in via AMQ. The index then exports parsed logs to ElasticSearch which provides analytics tools. Logstash additionally provides a web interface which communicates with ElasticSearch in order to analyse, retrieve and visualise log data.

LogStash is written in JRuby and communicates using AMQP with Redis serving as a message broker. Scalability is of chief concern to LogStash and it enables distribution of work via Redis, which despite relying on a centralised model can conceptually allow LogStash to scale to thousands of nodes.

MonALISA

MonALISA (Monitoring Agents in A Large Integrated Services Architecture) [52,53] is an agent based monitoring system for globally distributed grid systems. MonALISA uses a network of JINI [54] services to register and discover a variety of self-describing agent-based subsystems that dynamically collaborate to collect and analyse monitoring state. Agents can pull interact with conventional monitoring tools including Ganglia and Nagios and collect state from a range of applications in order to analyse and manage systems on a global scale.

MonALISA sees extensive use with the eScience and High Energy Physics community where large scale, globally distributed virtual organisations are common. The challenges those communities face are relatively unique, but few other monitoring systems address the concerns of widespread geographic distribution: a significant concern for cloud monitoring.

visPerf

visPerf [55] is a grid monitoring tool which provides scalable monitoring to large distributed server deployments. visPerf employs a hybrid peer-to-peer and centralised monitoring approach. Geographically close subsets of the grid utilise a unicast strategy to have a monitoring agent on each monitored server disseminate state to a local monitoring server. Each local server communicates with each other using a peer-to-peer protocol. A central server, the visPerf controller, collects state from each of the local masters and visualises state.

Communication in visPerf is facilitated by a bespoke protocol over TCP or alternatively XML-RPC. Either may be used allowing external tools to interact and obtain monitoring state. Each monitoring agent collects state using conventional UNIX tools including *iostat*, *vmstat* and *top*. Additionally, visPerf supports the collection of log data and uses a grammar based strategy to specify how to parse log data. The visPerf controller can communicate with monitoring agents in order to change the rate of data collection, what data is obtained and other variables. Bi-directional communication between monitoring agents and monitoring servers is an uncommon feature which is enables visPerf to adapt to changing monitoring requirements without reduced need to manually edit configuration.

GEMS

Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems (GEMS) is a tool designed for cluster and grid monitoring [5]. It's primary focus is failure detection at scale. GEMS is similar to Astrolabe in that it divides monitored nodes into a series of layers which form a tree. A gossip protocol [56] is employed to propagate resource and availability information throughout this hierarchy. A gossip message at level k encapsulates the state of all nodes at k and all levels above k. Layers are therefore formed based on the work being performed at each node, with more heavily loaded nodes being placed at the top of the hierarchy where less monitoring work is performed. This scheme is primarily used to propagate four data structures: a gossip list, suspect vector, suspect matrix and a live list. The gossip list contains the number of intervals since a heartbeat was received from each node, the suspect vector stores information regarding suspected failures, the suspect matrix is the total of all node's suspect vectors and the live list is a vector containing the availability of each node. This information is used to implement a consensus mechanism over the tree which corroborates missed heartbeats and failure suspicions to detect failure and network partitions.

This architecture can be extended to collect and propagate resource information. Nodes can operate a performance monitoring component which contains a series of sensors, small programs, which collect basic system metrics including cpu, memory, disk io and so forth and propagates them throughout the hierarchy. Additionally GEMS operates an API to allow this information to be made available to external management tools, middleware or other monitoring systems. These external systems are expected to analyse the collected monitoring state and take appropriate action, GEMS provides no mechanism for this. GEMS does however include significant room for extension and customisation by the end user. GEMS can collect and propagate any arbitrary state and has the capacity for real time programmatic reconfiguration.

Unlike Astrolabe, which GEMS bears similarity to, prototype code for the system is available [57]. This implementation has received a small scale, 150 node, evaluation which demonstrates the viability of the architecture and the viability of gossip protocols for monitoring at scale. Since its initial release, GEMS has not received widespread attention and has not been evaluated specifically for cloud monitoring. Despite this, the ideas expressed in GEMS are relevant to cloud computing. The problem domain for GEMS, large heterogeneous clusters are not entirely dissimilar to some cloud environments and the mechanism for programmatic reconfiguration is of definite relevance to cloud computing.

Reconnoiter

Reconnoiter [58] is an open source monitoring tool which integrates monitoring with trend analysis. Its design goal is to surpass the scalability of previous monitoring tools and cope with thousands of servers and hundreds of thousands of metrics. Reconnoiter uses a n-tier architecture similar to Nagios. The hierarchy consists of three components notid, stratcond and Reconnoiter. An instance of the monitoring agent, notid, runs in each server rack of datacenter (dependant upon scale) and performs monitoring checks on all local infrastructure. stratcond aggregates data from multiple notid instances (or other stratcond instances) and pushes aggregates to a PostgreSQL database. The front end, named Reconnoiter, performs various forms of trend analysis and visualises monitoring state.

Reconnoiter represents an incremental development from Nagios and other hierarchical server monitoring tools. It utilises a very similar architecture to previous tools but is specifically built for scale, with notid instances having an expected capacity of 100,000 service checks per minute. Despite being available for several years Reconnoiter has not seen mass adoption, possibly due to the increasing abandonment of centralised monitoring architectures in favour of decentralised alternatives.

Other monitoring systems

There are other related monitoring systems which are relevant to cloud monitoring but are either similar to the previously described systems, poorly documented or otherwise uncommon. These systems include:

- Icinga [59] is a fork of Nagios which was developed to overcome various issues in the Nagios development process and implement additional features. Icinga, by its very nature includes all of Nagios' features and additionally provides support for additional storage back ends, built in support for distributed monitoring, an SLA reporting mechanism and greater extensibility.
- Zenoss [60] is an opens source monitoring system similar in concept and use to Nagios. Zenoss include some functionality not present in a standard Nagios install including automatic host discovery, inventory and configuration management, and an extensive event management system. Zenoss provides a strong basis for customisation and extension providing numerous means to include additional functionality, including supporting the Nagios plugin format.
- Cacti [61], GroundWork [62], Munin [63], OpenNMS [64], Spiceworks [65], and Zabbix [66] are all enterprise monitoring solutions that have seen extensive use in conventional server monitoring which now see some usage in cloud monitoring.

These systems have similar functionality and utilise a similar set of backend tools including RRDTool, MySQL and PostgreSQL. These systems, like several discussed in this section, were designed for monitoring fixed server deployments and lack the supports for elasticity and scalability that more cloud-specific tools offer. The use of these tools in the cloud domain is unlikely to continue as alternative, better suited tools become increasingly available.

Cloud monitoring systems

Cloud monitoring systems are those systems which have been designed from the ground for monitoring cloud deployments. Typically such systems are aware of cloud concepts including elasticity, availability zones, VM provisioning and other cloud specific phenomena. Conceptually these tools have advantages over non cloud-aware tools as they can utilise cloud properties to scale and adapt as the system they monitor changes. The majority of systems in this category are recent developments and lack the user base, support and range of plugins that are available for many earlier systems. This presents a trade-off between cloud awareness and functionality that will likely lessen as time passes.

cloudinit.d

cloudinit.d [10] is a tool for launching and maintaining environments built on top of IaaS clouds. It is part of the Nimbus tool set which provides IaaS services for scientific users. Modelled after UNIX's init daemon, cloudinit.d can launch groups of VMs and services according to a set of run levels and manage the dependencies between those services. A set of config files known as a plan are used by cloudinit.d to describe a deployment and provide scripts to initialise services and to check for correct behaviour. Once cloudinit.d has launched the set of services and VMs specified in the plan it initiates monitoring. cloudinit.d leverages conventional UNIX tools in its monitoring system, using ssh and scp to deploy monitoring scripts from the plan on each VM. These scripts periodically check that status of the hosted service and push the result to the Nimbus front end. The main purpose of this monitoring scheme is to enable automated recovery. If a VM is detected as faulty by cloudinit.d it will terminate and swap out that VM with a new, hopefully correctly functioning VM. This simple recovery mechanism allows distributed services to keep running when VMs that make up part of the service fail.

The cloudinit.d documentation recommends integrating Nagios or Pingdom to provide more comprehensive and scalable monitoring services [67]. By itself cloudinit.d does not provide full monitoring services. Instead it only provides monitoring according to the scripts that are included within the plan. Failure or anomaly's behaviour which is not checked for by the monitoring scripts will go undetected. This impacts the failure recovery method as only simple failures which have been anticipated as potential failure can be detected. Unexpected failures which do not have scripts to detect them and failures that are not solved by swapping out VMs cannot be detected or recovered from. cloudinit.d is however notable as being one of the few monitoring tools which integrates into a deployment and management tool and is one of the few tools which include a failure recovery mechanism.

Sensu

Sensu [68] is described as a 'monitoring router' a system that takes the result of threshold checks and passes them to event handlers. Its primary goal is to provide an event based model for monitoring. In Sensu nomenclature, clients are monitored hosts that run small scripts for performing service checks. The server periodically polls clients who in turn execute their scripts and push the results to the server. The server then correlates and analyses the results of the service checks and upon any check exceeding a predefined threshold then calls handlers: user defined scripts which will attempt to take action.

Sensu makes use of RabbitMQ [69], a message orientated middleware platform based upon the AMQ standard. A Redis datastore is also used in order to store persistent data. These two technologies are used to allow Sensu to scale. Performance evaluation of RabbitMQ demonstrates its ability to handle tens of thousands of messages per second [70] and conceptually scale to tens of thousands of hosts.

Sensu is still an early stage project but already there are a number of third party check scripts and handlers available. With additional community adoption Sensu could become a predominant monitoring tool.

SQRT-C

In [71] An et al propose SQRT-C: a pubsub middleware scheme for real time resource monitoring of cloud deployments. They identify current monitoring tools reliance upon RESTful, HTTP and SOAP APIs as limitations to real time monitoring. They contend that protocols which are not, from the outset, designed for real time communication cannot sufficiently guarantee the delivery of time-sensitive monitoring state. To address these issues they propose SQRT-C: a real time resource monitoring tool based upon the OMG Data Distribution Service pubsub middleware. SQRT-C is based around three components: publishers, subscribers and a manager. All monitored nodes operate a publisher which publish state changes via the DDS protocol. A subscriber or set of subscribers run as close to the publishers as is feasible and analyse monitoring state and enact decisions in real time. The manager

orchestrates DDS connections between publisher and subscriber.

An implementation of SQRT-C is available for download. This implementation has been evaluated on a small testbed of 50 VMs. This number of VMs does not demonstrate the architectures viability for operating at scale, however the results do demonstrate the inherent QoS improvements that the architecture brings. By utilising an underlying protocol that provides strong QoS guarantees SQRT-C demonstrates significantly reduced latency and jitter compared to monitoring tools leveraging conventional RESTful APIs.

SQRT-C has not received particularly widespread attention but the issues of real time monitoring remain unaddressed by monitoring tools at large. Virtually all cloud monitoring tools expose data via a conventional HTTP API often with no alternative, the HP Cloud Monitoring tool is the only notable monitoring system which offers a real time alternative.

Konig et al

In [7] Konig et al. propose a distributed peer to peer tool for monitoring cloud infrastructure. [7] is a three level architecture consisting of a data, processing and distribution layer. The data layer is responsible for obtaining monitoring data from a range of sources including raw log files, databases and services in addition to other conventional monitoring tools including Ganglia and Nagios. The processing layer exposes a SQL-like query language to filter monitoring data and alter system configuration at run time. The distribution layer operates a peer to peer overlay based on SmartFrog [72] which distributes data and queries across the deployment.

Konig et al., 2012 [7] is similar to other p2p monitoring systems in terms of topology but introduces a powerful query language for obtaining monitoring data as opposed to a conventional, simple REST API. Additionally the data layer component abstracts over numerous data sources, including other monitoring systems making this work a notable tool for building federated clouds and other architectures which encapsulate existing systems.

Dhingra et al

In [73] Dhingra et al. propose a distributed cloud monitoring framework which obtains metrics from both the VM and underlying physical hosts. Dhingra et al. contend that no current monitoring solutions provide 'customer focused' monitoring and fail to monitor phenomena at and below the hypervisor level. Their proposed architecture, which is conceptually similar in design to collectd, runs monitoring agents on both the VM and physical host and aggregates state at a front end. The front end module correlates VM level metrics with low level physical metrics to provide a comprehensive monitoring data to the end user. Additionally they propose a mechanism for adjusting the level of granularity that users can receive in order to provide fine grain data when users are performing their own analysis and decision making or more coarse grain data when users are relying upon the cloud provider to make decisions on their behalf. No implantation of the proposed architecture is available for download, however the variable granularity monitoring an multi-level monitoring present novel concepts which have not yet been fully integrated to existing monitoring tools.

DARGOS

Distributed Architecture for Resource manaGement and mOnitoring in cloudS (DARGOS)[6] is a fully decentralised resource monitor. DARGOS, like SQRT-C, makes use of the OMG Data Distribution Standard [74] (DDS) to provide a QoS sensitive pubsub architecture. DARGOS has two entities: the Node Monitor Agent (NMA) and the Node Supervisor Agent (NSA) which for all intents and purposes are the publisher and subscriber, respectively. The NMA collects state from the local VM and publishes state using the DDS. The NSA are responsible for subscribing to pertinent monitoring state and making that state available to analysis software, users or other agents via an API or other channel. DARGOS includes two mechanisms to reduce the volume of unneeded monitoring state which is propagated to consumers: time and volume based filters. These mechanisms reduce unnecessary traffic and yield improvements over other pubsub schemes.

While DARGOS makes use of the DDS standard, QoS is not its primary concern and no specific provisions are made to ensure the low latency and jitter achieved by SQRT-C. Conceptually, DARGOS is similar to other pubsub based monitoring tools including GMOnE, SQRT-C and Lattice

CloudSense

CloudSense [75] is a data centre monitoring tool which, unlike other tools surveyed here, operates at the switching level. CloudSense attempts to address networking limitations in previous tools which prevent the collection of large volumes of fine grain monitoring information. CloudSense operates on switches and makes use of compressive sensing [76], a recent signal processing technique which allows for distributed compression without coordination, to compress the stream of monitoring state in the network. This allows CloudSense to collect greater monitoring information without using additional bandwidth. In the proposed scheme each rack switch in a datacenter collects monitoring state from servers within each rack and compresses and transmits aggregated state to a master server which detects anomalous state.

CloudSense is primarily intended for datacenter monitoring and uses a compression scheme which lends itself well to anomaly detection. CloudSense has been proposed as a tool for monitoring MapReduce and other performance bound applications where anomaly detection is beneficial. This architecture in itself, is not enough to provide a comprehensive monitoring solution but does present a novel option for anomaly detection as part of a larger tool.

GMonE

GMonE [77] is a monitoring tool which covers the physical, infastructure, platform and application layers of the cloud stack. GMonE attempts to overcome the limitations of existing monitoring tools which give only a partial view of a cloud system by giving a comprehensive view of all layers of the cloud. To this end, a monitoring agent: GMonEMon is present in all monitored components of the cloud stack including physical servers, IaaS VMs, PaaS instances and SaaS apps. GMonEMon has a plugin architecture that allows users to develop additional modules to obtain metrics from all relevant sources. GMonEMon uses its plugins to collect data and then uses Java RMI based publish subscribe middleware to publish monitoring state. A database component: GMonEDB acts as a subscriber to various GMonEMon instances and stores monitoring state using MySQL, RRDtool, Cassandra [78] or other back end.

Unlike the vast majority of monitoring tools surveyed here, GMondE provides monitoring services to both cloud providers and cloud users. In a GmonE deployment there are at least two GMonE DB instances: one for a user and one for the provider. Each stakeholders subscribe to the components relevant to their operations. This could be used to allow users to obtain monitoring state from the layers bellow the layer that they are operating on. For example as SaaS could use this scheme to obtain metrics regarding the PaaS platform, the VMs running the platform and even potentially the underlying hardware. The provider, meanwhile could choose only to monitoring physical resources or could monitor their users infrastructure.

GMonE is a recent monitoring tool which currently lacks any publicly available release. Despite this, GMonE's multi layer monitoring solution and novel pubsub scheme are notable concepts which are not found in other contemporary monitoring tools.

Lattice

Lattice [79] is a monitoring platform for virtual resources which attempts to overcome the limitations of previous monitoring tools, including Ganglia and Nagios, that do not address elasticity, dynamism and frequent change. Unlike other tools in this survey Lattice is not in itself a monitoring tool, rather it is a platform for developing monitoring tools. Lattice includes abstractions which are similar to other monitoring tools: producers, consumers and probes. Additionally Lattice provides the concept of a data source and distribution framework. The data source is an producer which encapsulates the logic for controlling the collection and transmission of monitoring data in a series of probes which perform the actual data collection. The distribution framework is the mechanism which transmits monitoring state between the producers and the consumers.

Lattice does not provide full implementations of these structures but rather provides building blocks from which third parties can develop full monitoring solution. This design is intended to allow developers to build monitoring solutions specific to their unique use cases. The separation of concerns between the various monitoring components allows components to be differently implemented and change over time without affecting the other components.

The notion of a framework for building monitoring tools is a novel break from the other tools surveyed in this paper. Conceptually Lattice can be used to build different monitoring tools for different use cases rather than reapplying existing tools to different uses cases. While this allow for the best fitting tools possible it requires significant labour to develop and test tools based upon Lattice. Lattice does not provide a library of probes, requiring the developer to implement their own library of data collection scripts, a significant limitation when compared to other tools including collectd and Nagios. Additionally, Lattice requires the developer to make design decisions regarding the distribution framework; which network architectures, wire formats, discovery mechanisms and so forth are used. This degree of effort is likely to be prohibitive to the vast majority of users.

Lattice being a framework for developing monitoring tools and not a tool in itself does not merit a direct classification and is capable of producing tools that fit every classification.

OpenNebula monitoring

OpenNebula [80] is an open source toolkit for building IaaS clouds which includes a monitoring system [81,82]. OpenNebula manages the VM lifecycle in a manner similar to cloudinit.d, it bootstraps the VM with the necessary monitoring agent and small scripts called probes via SSH. OpenNebula has two configurations: pull and push. The push model is the preferred mode of operation. In this configuration the OpenNebula monitoring agent collects resource metrics and transmits them to the OpenNebula front end via UDP. The front end operates a collection daemon which receives monitoring state and periodically sends batches to oned, the OpenNebula core daemon.

If, due to support issues, the push model is unavailable OpenNebula defaults to a pull model, similar to Nagios, whereby oned initiates an SSH connection on each monitored VM, executes the probes and pulls the results to the front end. Visualisation and alerts can then be generated by the OpenNebula web front end. OpenNebula's monitoring system is not particularly novel but is one of the few examples of an open source monitoring system which is embedded within a cloud infrastructure.

PCMONS

Private Clouds MONitoring Systems (PCMONS) [83] is a monitoring tool which was designed to address the lack of effective open source tools for private cloud monitoring. PCMONS employs a three layer structure consisting of the view, integration and infrastructure layer. The infrastructure layer is comprised of heterogeneous resources: IaaS clouds, clusters or other server deployments. In the release available for download [84] Eucalyptus [85] and OpenNebula are supported at the infrastructure level. The integration level acts as a translator which abstracts the heterogeneity of the infrastructure level presenting a uniform view of otherwise disparate systems. The integration layer is responsible for generating the required configuration and installing the necessary software to collect monitoring data from the infrastructure level and passing it in a form that the view layer can use. The view layer performs the visualization and analysis of monitoring data. In the release of PCMONS, the view layer is based upon the Nagios server.

The most novel feature of PCMONS is the notion of the integration layer, a concept which is essential for federated cloud monitoring whereby monitoring is performed over a series of different cloud resources. This feature of PCMONS is found in few other monitoring tools surveyed here.

Varanus

Varanus [86] is a peer to peer monitoring tool designed for monitoring large scale cloud deployments. Designed to handle scale and elasticity, Varanus makes use of a k-nearest neighbour based group mechanism to dynamically group related VMs and form a layered gossip hierarchy for propagating monitoring state. Varanus uses three abstractions: groups, regions and cloud over which the gossip layer of the hierarchy. Groups are collections of VMs assigned by Varanus' grouping strategy, regions are physical locations: data centres, cloud regions or similar, and the cloud is the top level abstraction including one or more regions. At the group and region level Varanus makes heavy use of the network to propagate state and state aggregates to related VMs. At the cloud level communication is slower and only periodic samples are propagated. This scheme attempts to exploit the inherent bandwidth differences in and between cloud regions. Heavy use of the network is used where bandwidth is unmetered and fast and reduced network usage occurs between regions where bandwidth is metered and limited.

Varanus has some points of commonality with Astrolabe and GEMS in that it utilises a gossip hierarchy to propagate monitoring state but introduces applies these concepts in a manner more suitable to cloud monitoring.

Monitoring as a service tools

The following systems are designed from the outset to monitoring cloud deployments. Unlike the systems detailed in the previous section their design is not published nor are any implementations available for full evaluation. Instead, these systems are monitoring as as a service (MaaS) tools which are accessible only though an API or other interface.

As the backend portions of monitoring as a service tools are hidden from the end user it is difficult to classify these tools. Externally, each of these tools are unicast push tools. The architecture used by the provider is, however, unknown.

Amazon CloudWatch

CloudWatch [87] is the monitoring component of Amazon Web Services. CloudWatch primarily acts as a store for monitoring data, allowing EC2 instances and other AWS services to push state to it via an HTTP API. Using this data a user can view plots, trends, statistics and various other representations via the AWS management console. This information can then be used to create alarms which trigger user alerts or autoscale deployments. Monitoring state can also be pulled by third party applications for analysis or long term storage. Various tools including Nagios have support for obtaining CloudWatch metrics.

Access to this service is governed by a pricing model that charges for metrics, alarms, API requests and monitoring frequency. The most significant basic charge is for metrics to be collected at minute intervals followed by the charge for the use of non standard metrics. CloudWatch presents a trade-off between full customisability and ease of use. The primary use case of CloudWatch is monitoring the full gamut of AWS services. Users of only EC2 will likely find the customisability of a full monitoring system preferable to the limited control afforded by CloudWatch.

HP cloud monitoring

HP Cloud Monitoring [88] is the monitoring component of the HP Public Cloud [89] which is currently in public beta. HP Cloud Monitoring has features that are equivalent to CloudWatch, offering threshold based alarms, alerting and data visualization. Unlike, CloudWatch, HP monitoring places emphasis on real time monitoring and

provides a high-performance message queue endpoint in addition to a REST API in order to transmit large volumes of monitoring state to third party tools in real time. Thus, unlike most other MaaS tools, HP cloud monitoring is intended to support rather than supplant other monitoring tools. Being able to use both a monitoring service and a full fledged monitoring system potentially offers the best of both worlds making this a notable system.

RightScale monitoring system

Rightscale [40] is a multi-cloud cloud management service, the payed edition of which, includes the RightScale Monitoring System [90]. The monitoring system is a collectd based stack which is integrated with RightScale's management tools. In a manner similar to cloudinit.d, Rightscale manages service deployment and deploys scripts to newly instantiated VMs to provide various management functions, including monitoring. These scripts deploy collectd and supporting tools which collects and transmits monitoring state to a Rightscale server at 20 seconds intervals via UDP unicast. The monitoring server stores data in a RRDtool database and the end user can view this data using a proprietary dashboard interface. Alerts, graphs and raw data can be obtained via a REST API.

Rightscale's configuration collectd operates over simple unicast and there is relatively limited analysis and complex visualisation available from the dashboard. For more complex analysis and visualisation, third party tools are required. Despite the limitations, the availability of a managed collectd stack makes Rightscale monitoring a notable tool.

Rightscale's monitoring system takes away much of the difficulty of rolling your own collectd based monitoring a notable tool.

New relic

New Replic [91] provides a set of SaaS monitoring and analytics tools for a range of services including servers and applications. New Relic uses a locally deployed monitoring agent to push state to a centralised dashboard which performs a comprehensive set of time series and other visualizations. New Relic places focus upon web applications providing an analysis of of application performance, response time, requests per minute and other phenomena. With regards to server monitoring New Relic provides a less comprehensive set of resource metrics, primary: CPU, memory and disk usage.

New Relic has partnerships with AWS, Azure, Rackspace and other major cloud providers and provide an support the monitoring of most major cloud services. The most notable features of New Relic is it's user interface which provides an extensive set of visualisations and reporting.

CopperEgg

CopperEgg [92] is a monitoring as a service tool which provides server, EC2, database and website monitoring. CopperEgg is similar to New Relic and other SaaS monitoring tools in that it utilises a small monitoring agent to push state from monitored hosts to a service endpoint. CopperEggs most notable unique feature is integration with numerous current tools including Redis, MongoDB, Chef, PostresQL, MySQL and Apache. Thus, unlike New Relic, CopperEgg is capable of providing a significant range of application monitoring metrics in addition to more basic resource usage metrics. CopperEgg provides an intuitive web front end for interrogating monitoring data which surpasses the available interfaces of most non SaaS tools.

Additional services

There are an extensive number of monitoring as a service tools, each with a slightly different feature set. These tools include:

- Cloud Sleuth [93] is a tool designed to monitoring service availability and performance.
- Montis [94] is an agent less monitoring service which polls monitored host and services from numerous locations to achieve a global view of availability and performance.
- Stackdriver [95] is an intelligent monitoring tools for AWS, Google Compute Engine and Rackspace Cloud that provides resource monitoring and anomaly detection.
- Boundry [96] is a monitoring aggregator that can consumer data from other tools including: CloudWatch, Splunk, Chef, Nagios, Zenoss and others.
- Cloudyn [97] is a tool focussed on providing cost usage analysis for AWS and Google Compute Engine.

Taxonomy

From our extensive survey of existing monitoring systems it is evident that there are a number of common elements. These elements include how components are architected and how they communicate, the motivation or origin behind the tool and the tool's primary use case. These can be used to form a taxonomy to classify current and future monitoring tools. Figure 3 illustrates this taxonomy.

Architecture, communication and collection

Monitoring frameworks typically consist of a number of monitoring agents – components responsible for the monitoring process. The way in which these agents are architected, communicate and collect data differs from system to system, however a number of common patterns

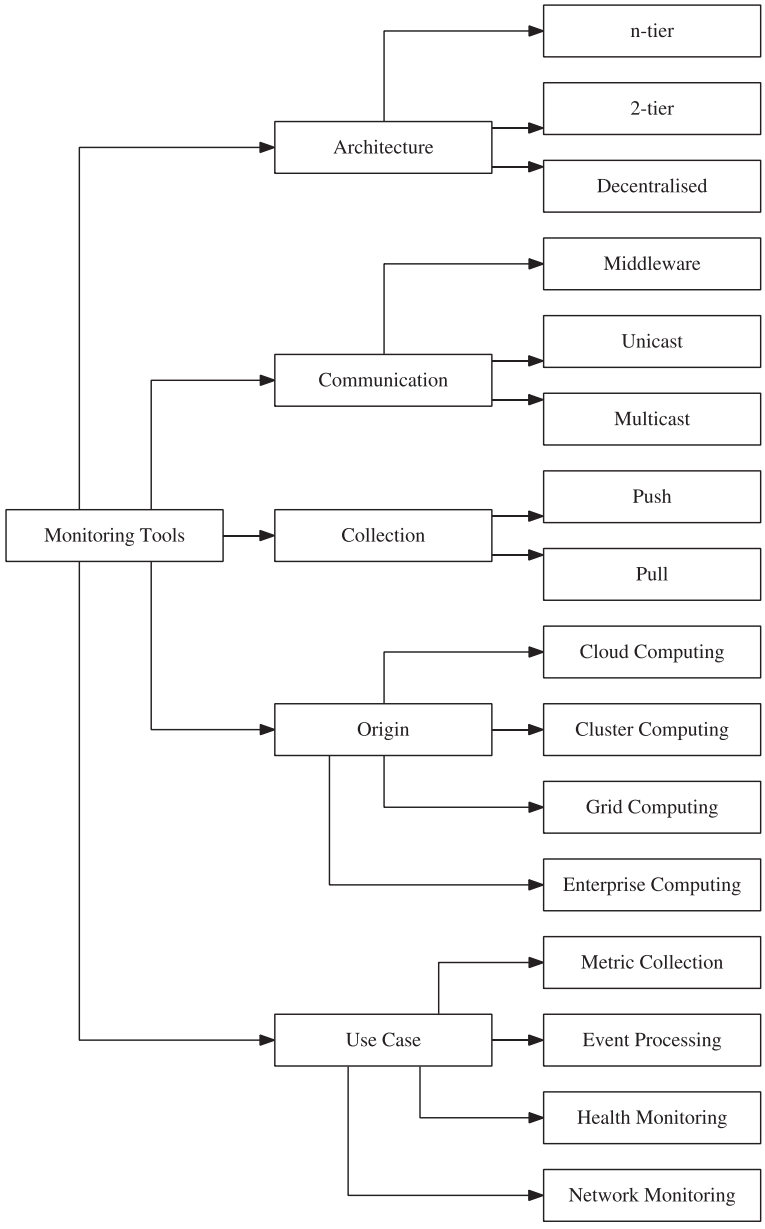

Figure 3 Taxonomy of Monitoring Tools: each tool is classified through an architecture, communication mechanism, collection mechanism, origin and use-case.

can be identified, which forms part of our taxonomy. Monitoring agents are typically architected in one of the following ways:

- Two tier: in the most basic monitoring architecture there are two types monitoring agents: a producer and a consumer. Monitored hosts run an agent simply comprising a producer. A second type of agent comprising the consumer, state store and front end collects, stores and analyses state from the first type of agent. This architecture is shown in Figure 4.
- N-tier: this scheme is an evolution of the previous scheme. The two agents described in the previous section remain and a third agent is introduced. A new intermediate agent acts as a consumer, state store and republisher. The intermediate agent collects state either from producer agents or from other intermediate agents. The consumer agent consumes state from intermediate agents. The n-tier architecture is illustrated in Figure 5.
- Decentralised: all monitoring agents have the same components. Agents are all capable of the same

functionality and their exact operation is dependant upon locality, behaviour or other factors at runtime. The decentralised architecture is shown in Figure 6.

Monitoring agents are then connected by one of the following communication channels:

- Unicast: monitoring agents communicate according to a simple unicast protocol.
- Multicast: agents communicate with groups rather than distinct agents. This includes the use of IP multicast and overlay based multicast.
- Middleware: communication between agents is facilitated via a middleware application. This includes publish-subscribe middleware, message queues, message brokers, service buses or other OS independent messaging.

A final point of consideration is with regards to data collection. This pertains to the directionality of the communication channel. There are two contrasting mechanism for collecting data from monitored hosts: push and pull. In pull systems the consumer initiates the data collection transaction. In push systems the producer initiates the transaction.

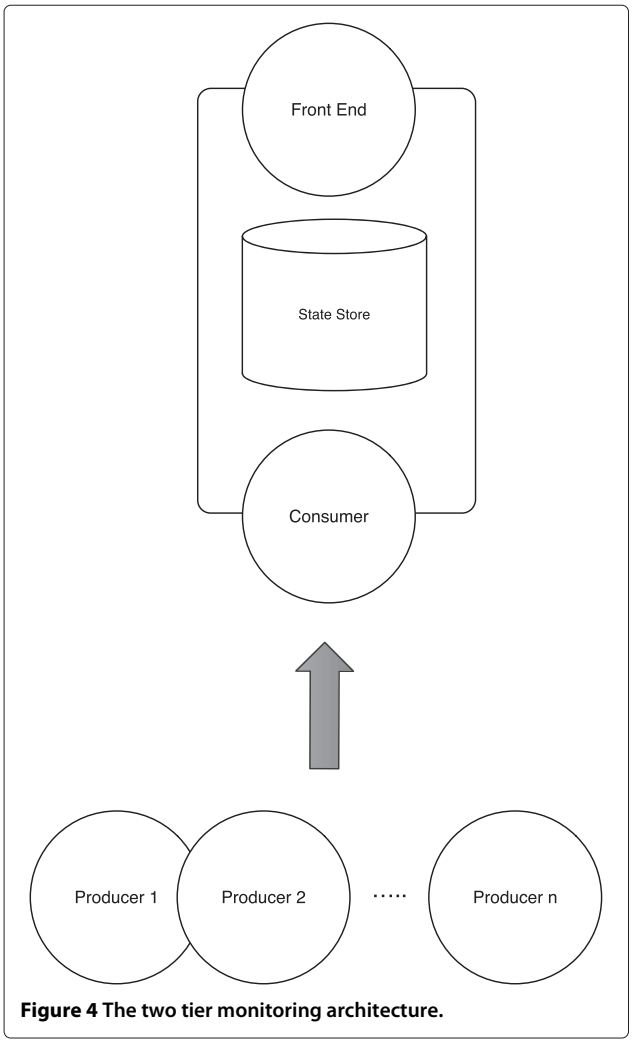

Figure 4 The two tier monitoring architecture.

Origin

The initial developer of a tool is often not the developer who goes on to maintain and extend that tool. Never the less, the intention and motivation behind the original development does give some indication as to the design of the monitoring tool and its applicability to cloud monitoring. In our survey there are four origins: cluster/hpc, grid, cloud and enterprise computing. Cluster monitoring tools were predominantly written to operate over a data centre or other small geographic area and were envisaged to function in an environment with few applications. As such they tend to focus primarily upon metric collection and little else. Ganglia is a prime example of a cluster monitoring tool. Grid tools are a natural evolution of cluster tools whereby monitoring is performed over a wider geographic area as is typical of grid virtual organisations. In a similar fashion, grid tools focus primarily on metric collection though health monitoring also features in several grid based tools. Enterprise monitoring is a vast category of tools which are incorporate several use cases. Enterprise monitoring tools are by in large designed for organisations who run a variety of applications, operating systems, hardware and other infrastructure. Enterprise tools such as Nagios are commonly used in cloud settings as they have wide support for applications such as web servers, databases and message queues. Enterprise monitoring tools were not, largely, designed to tolerate scale or changes to scale. Such tools therefore often require manual configuration to add or remove a monitored host and incur heavy load when monitoring large numbers of VMs. Cloud monitoring tools are the newest category of monitoring tools. These tools are in their infancy but are typically designed with scale and elasticity as core goals. Cloud monitoring tools are often interoperable and represent the growing popularity of patchwork solutions which integrate numerous tools. While, a priori, cloud monitoring tools are most appropriate for cloud monitoring they often lack the features of their more mature counterparts from alternative domains.

Use case

Monitoring is an all encompassing umbrella term for a number of distinct processes. While some tools attempt to provide extensive functionality which covers multiple use cases, the vast majority of tools cater for a single pertinent use case. These use cases include: metric collection, log/event processing, health monitoring and network monitoring.

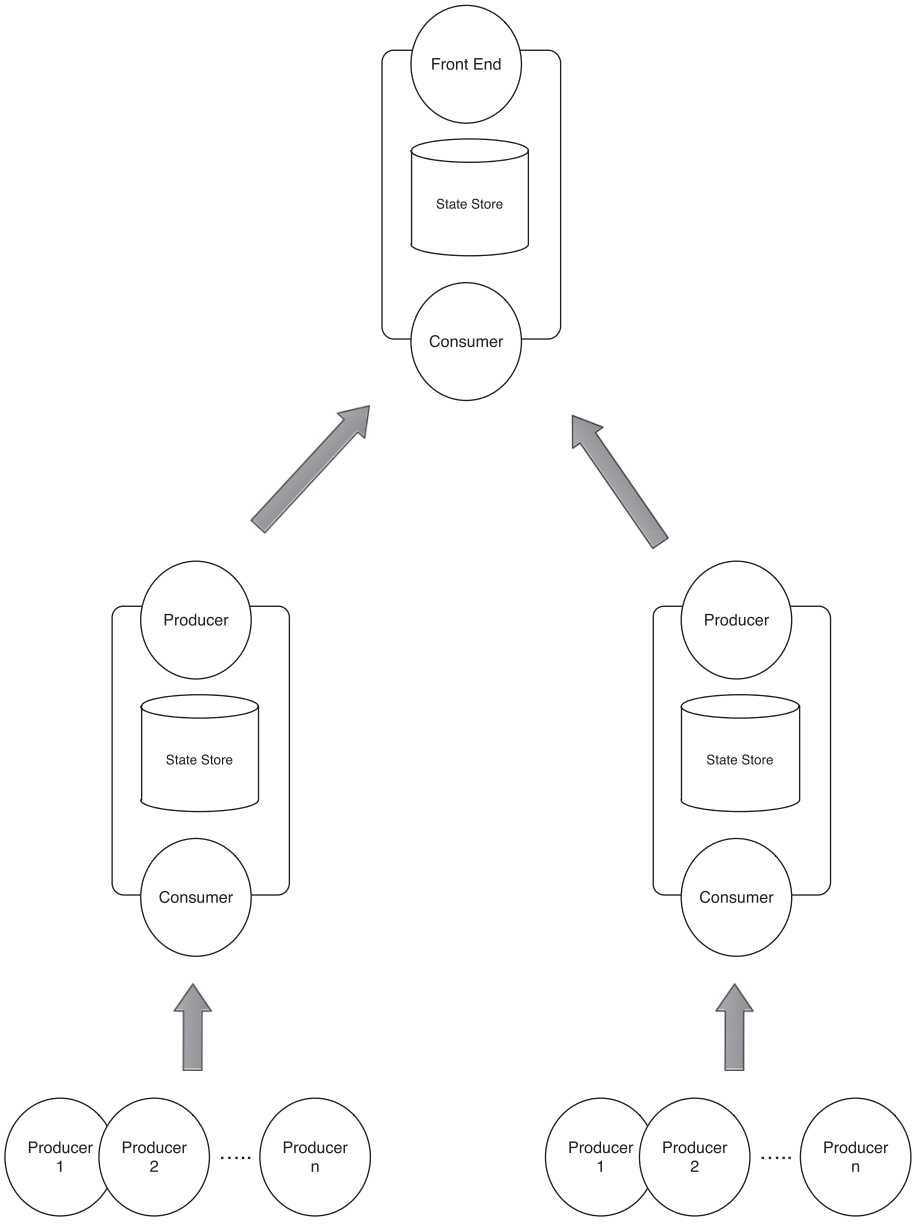

Figure 5 The n-tier monitoring architecture.

Metric collection is amongst the most basic monitoring operation. It involves the collection of statistics regarding system performance and utilisation and typically yields a series of graphs as an end product. Metric collection tools are typically less complex than other tools and do not typically involve complex analysis. Tools which are indented for metric collection include Ganglia, Graphite and collectd and StatsD.

Log and event processing is a more complex use case and typically involves an API or other interface which allows a user to define a pipeline or set of instructions to process a stream of events. Event processing is a much more expensive form of monitoring than alternative use cases and requires significant compute capacity to keep up with the demands of real time stream processing. Event processing tools include Riemann and Logstash. Health monitoring is similar to metric processing in that it typically yields a series of values representing the status of a service. It differs from more simple metric collection in that it typically involves interaction with services. A common example of this is Apache health monitoring whereby tools communicate with the Apache web server via *mod_status*, *apachectl* or via http in order to obtain internal state. This use case is

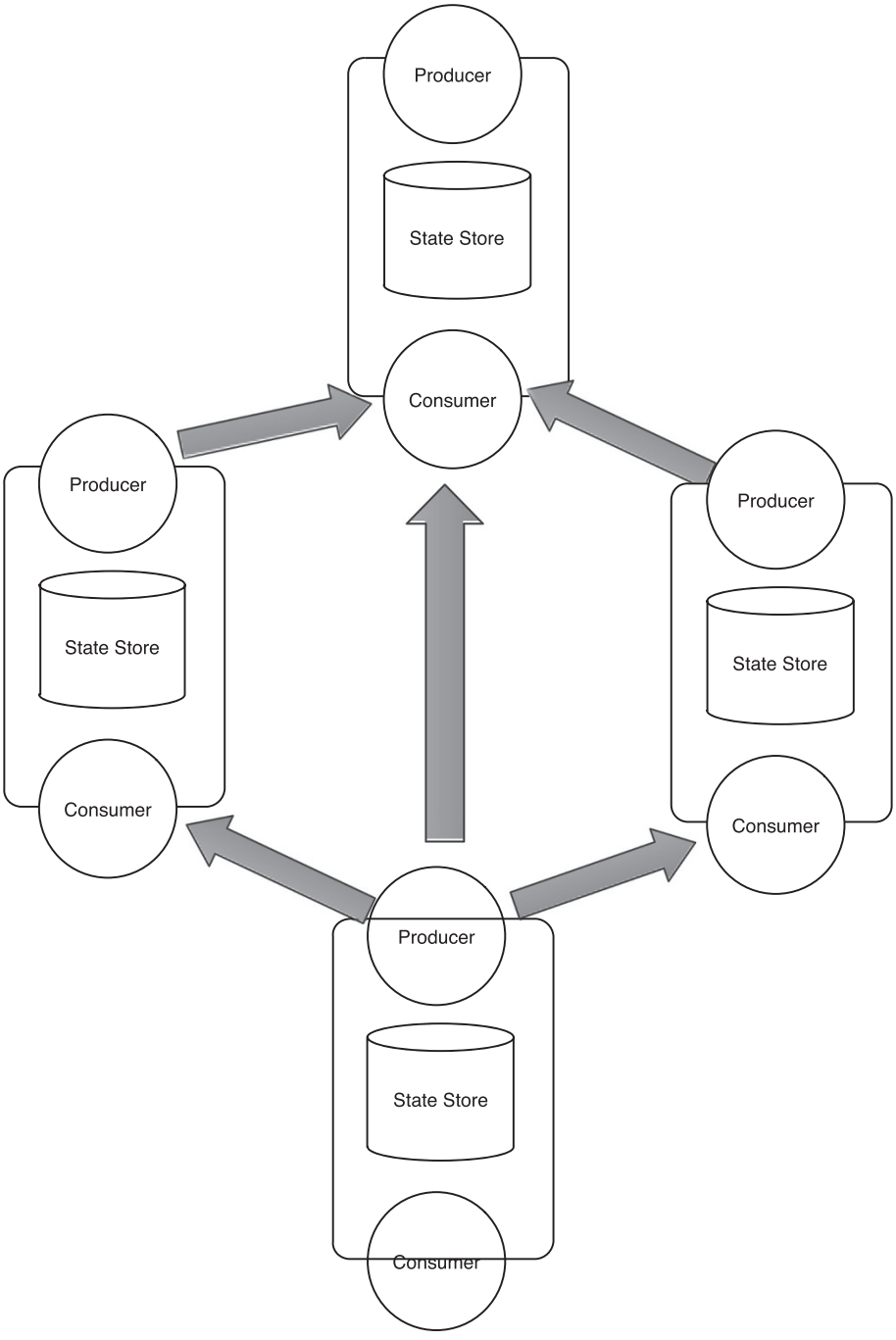

Figure 6 The decentralised monitoring architecture.

most commonly associated with alerting and reactive monitoring.

Health monitoring often requires the writing of bespoke software and frequently relies on plugin and extension mechanisms exposed by monitoring tools. Nagios, Zabbix and Icinga all cater to this use case. Network monitoring is a wide use case which includes the monitoring of network performance, network devices and network services. This use case overlaps with health monitoring but focuses more upon network conditions opposed to simply the services available on that network. This often includes DoS detection, routing health detection, network partition detection and so forth. Tools which cater to network monitoring include Cacti, mrtg and ntop.

Applying the taxonomy

Table 1 provides a summary of the systems surveyed within the previous sections. When collated together, a trend emerges from the properties of the monitoring systems surveyed in previous sections. Older tools originating from the domain of enterprise monitoring are predominantly 2-tier tools which have since been modified to support n-tier architectures in order to improve scalability. Unexpectedly, many of the new generation of cloud monitoring tools are also based upon 2-tier

Table 1 Taxonomy of monitoring tools

System	Architecture	Communication	Collection	Origin	Use case
Astrolabe [8]	Decentralised	Multicast	Push	Cluster Computing	Metric Collection and Health Monitoring
Cacti [61]	2-tier	Unicast	Pull	Enterprise Computing	Network Monitoring and Metric Collection
cloudinit.d [10]	2-tier	Unicast	Push	Cloud Computing	Health Monitoring
CloudSense [75]	n-tier	Unicast	Push	Cloud Computing	Metric Collection
collectd [39]	2-tier/n-tier	Unicast/multicast	Push	Enterprise Computing	Metric Collection
DARGOS [6]	2-tier	Middleware	Push	Cloud Computing	Metric Collection
Dhingra et al [73]	2-tier	Unicast	Push	Cloud Computing	Metric Collection
Ganglia [31]	n-tier	Unicast	Push	Grid Computing	Metric Collection
GEMS [5]	Decentralised	Multicast	Push	Cluster Computing	Health Monitoring
GMonE [77]	2-tier	Middlewear	Push	Cloud Computing	Metric Collection
GroundWork [62]	n-tier	Unicast	Pull	Enterprise Computing	Metric Collection and Health Monitoring
Icinga [59]	n-tier	Unicast	Push	Enterprise Computing	Metric Collection and Health Monitoring
Konig et al [7]	Decentralised	Multicast	Push	Cloud Computing	Metric Collection
Logstash [41]	Centralised	Unicast	Push	Cloud Computing	Event Processing
MonALISA [52]	n-tier	Unicast	Push	Grid Computing	Metric Collection and Health Monitoring
Munin [63]	n-tier	Unicast	Pull	Enterprise Computing	Network Monitoring
Nagios [35]	2-tier/n-tier	Unicast	Pull	Enterprise Computing	Network Monitoring and Health Monitoring
OpenNMS [64]	2-tier/n-tier	Unicast	Pull	Enterprise Computing	Metric Collection and Health Monitoring
OpenNebula [10][80]	2-tier	Unicast	Push	Cloud Computing	Metric Collection
PCMONS [83]	2-tier	Unicast	Push	Cloud Computing	Metic Collection
Reconnoiter [5]	n-tier	Unicast	Push	Enterprise Computing	Metric Collection and Health Monitoring
Riemann [9]	2-tier	Unicast	Push	Cloud Computing	Event Processing
sFlow [48]	2-tier	Unicast	Push	Enterprise Computing	Network Monitoring
Spiceworks [65]	2-tier/n-tier	Unicast	Pull	Enterprise Computing	Network Monitoring
StatsD [42]	2-tier	Unicast	Push	Enterprise Computing	Metric Collection
SQRT-C [71]	2-tier	Middleware	Push	Cloud Computing	Metric Collection
Varanus [86]	Decentralised	Multicast	Push	Cloud Computing	Metric Collection
visPerf [55]	Centralised/decentralised	Unicast	Push	Grid Monitoring	Metric Collection and Log Processing
Zabbix [66]	2-tier/n-tier	Unicast	Pull	Enterprise Monitoring	Health Monitoring and Metric Collection
Zenoss [60]	2-tier/n-tier	Unicast	Pull	Enterprise Computing	Health Monitoring and Metric Collection

architectures. Many of the research projects which have investigated issues in cloud monitoring have examined issues other than architecture scalability and thus for the sake of simplicity or other concerns utilise 2-tier architectures. Those research projects which have placed scalability amongst their primary concerns have predominantly employed n-tier or decentralised architectures in addition to utilising some form of multicast or middleware to facilitate communication.

Another trend which is clear from Figure 4 is the shift from tools utilising a pull mechanism for collecting monitoring data to using a push model. This represents a move from tight coupling and manual configuration to loosely coupled tools which perform auto discovery and handle membership change gracefully. Pull mechanism provided a means for centralised monitoring servers to control the rate at which data was collected and control the volume of information produced at any given time. Push mechanism dispense with that means of control and either require an additional feedback mechanism to reintroduce this functionality or require the monitoring system to cope with variable and uncontrolled rates of monitoring data.

Applying the cloud monitoring requirements

Tables 2 and 3 summarise the cloud monitoring requirements (detailed in Section 1) which each of the surveyed monitoring systems provide. It is clear from these tables that non cloud specific tools, expectedly, implement less of the requirements of cloud monitoring than tools specifically built for cloud monitoring. Notably, the general monitoring systems do implement the comprehensiveness requirement more so than current cloud tools. This is primarily due to general tools having a greater install base, larger communities and have benefited from more development time. As cloud monitoring tools mature this disparity is likely to diminish. Autonomic monitoring features are the rarest of the requirements as much of the research in the field of autonomic systems is yet to be applied to monitoring. Notably, time sensitivity is not a feature provided by any general monitoring systems. Recent cloud monitoring tools, however, have begun to implement real time and time sensitive mechanisms which may begin to become common place.

The tools which support the less common monitoring requirements are predominantly academic proof of concept systems which do not have freely available releases or have unmaintained releases. As is common, there is likely to be a trickle down effect with the concepts introduced by academic proof of concept systems being adopted by open source and industrial tools. Notably, loose coupling and related fault tolerance mechanism which were once the preserve of academic research are now featuring in more mainstream monitoring tools. This trend is likely to follow for other cloud monitoring requirements.

A possible outcome of the increasing complexity of the cloud computing sector is the demise of all-in-one monitoring tools. Many of the tools surveyed here are not

Table 2 Requirements fulfilled by general monitoring systems

		Cloud monitoring requirements						
		Scalable	**Cloud aware**	**Fault tolerant**	**Multiply granular**	**Comprehensive**	**Time sensitive**	**Autonomic**
Monitoring systems	Astrolabe	✓		✓				
	Cacti					✓		
	collectd	✓				✓		
	Ganglia	✓		✓				
	GEMS	✓		✓				
	Icinga					✓		
	MonaLISA	✓				✓		
	Nagios					✓		
	OpenNMS					✓		
	Reconnoiter	✓						
	Riemann	✓						✓
	StatsD	✓			✓			
	sFLow					✓		
	visPerf	✓		✓	✓			
	Zabix					✓		
	Zenoss					✓		

Table 3 Requirements fulfilled by cloud monitoring systems

		Cloud monitoring requirements						
		Scalable	**Cloud aware**	**Fault tolerant**	**Multiply granular**	**Comprehensive**	**Time sensitive**	**Autonomic**
Monitoring systems	cloudinit.d		✓	✓				
	CloudSense	✓					✓	
	DARGOS	✓	✓		✓		✓	
	Dingra et al [73]		✓		✓			
	GMonE	✓	✓			✓		
	Konig et al [7]	✓	✓					
	Logstash	✓			✓		✓	
	OpenNebula		✓					
	PCMONS	✓	✓					
	Sensu	✓						✓
	SQRT-C	✓	✓				✓	
	Varanus	✓	✓				✓	

complete solutions, rather they provide part of the monitoring process. With the diversification of cloud providers, APIs, applications and other factors it will become increasingly difficult to develop tools with encompass all areas of the cloud computing stack. It is therefore likely that future monitoring solutions will be comprised of several tools which can be integrated and alternated to provide comprehensive monitoring. This trend is already coming to fruition in the open source domain where collectd, statsd, Graphite, Riemann and a variety of other tools which have common interchange formats can be orchestrated to provide comprehensive monitoring. This trend is slowly gaining traction amongst industrial tools such as many of the monitoring as a service tools which provide interoperability and complimenting feature sets. CopperEgg for example, is interoperable with a number of data sources including Amazon CloudWatch and the two tools provide contrasting feature sets and different levels of granularity. The rise of a 'no silver bullet' mindset would hasten the long predicted demise of conventional enterprise monitoring tools and see significant diversification of the cloud monitoring sector.

Monitoring as an engineering practice

The greatest extent of this survey is spent discussing the design of current monitoring tools in order to enable users to more effectively choose tools and for researchers to design more effective monitoring tools. Monitoring tools are not however the be all and end all of monitoring, they are however just one part of the process. Just as software engineering prescribes a length requirements engineering and design process prior to implemented so to does monitoring require a well thought out strategy. Monitoring tools form the bulk of the implementation of any strategy but without appropriate consideration the haphazard application of monitoring tools will inevitably fail to perform as required.

Monitoring strategies

A monitoring tool alone is not a monitoring strategy. Even the most sophisticated monitoring tools are not the be all and end all of monitoring, but rather are a part of a grander strategy. A monitoring strategy defines what variables and events should be monitored, which tools are used, who is involved and what actions are taken. As such, a monitoring strategy is a sociotechnical process which involves both software and human actors.

It is extremely difficult to produce an effective monitoring strategy and it is doubly difficult to produce a strategy before a system is operational. Most monitoring strategies are devised during the design stage but are constantly revised when events show the current strategy to be incomplete, inefficient or otherwise ineffective. Part of the inherent difficulty in devising a comprehensive monitoring strategy is the complex intertwinement of software and services within any large system. It is not immediately clear as to what effect various failure modes will have on other services. Often failure cascades and knock-on effects are extremely difficult to predict ahead of time. As such it may take numerous iterations for a monitoring strategy to reach a point whereby it can be used to prevent widespread failure or other issues arising. Many high profile outages which have affected large swathes of the web have been due to ineffective monitoring strategies which have prevented operations staff detecting and resolving a growing issue before it was too late.

Building a monitoring strategy first and foremost requires considering what actors (both human and software) will be involved in the monitoring process. Common examples of these include:

- Operations staff who are actively involved in maintaining the system and utilising monitoring data as a key part of their role.
- Other staff who use monitoring data to varying degrees within their roles.
- Monitoring tools which collect, analyse and visualise data from the system in question.
- The infrastructure provider, this could be a cloud provider, datacenter, an in-house team or a third party organisation.
- Software systems with produce and/or consume monitoring data.
- Customers or clients who may generate and make use of monitoring data.

This is by no means and extensive list but it represents many of the common actors involve in the monitoring process. Failure to consider the relevant actors can lead to ineffective monitoring strategy and can often result in "Shadow IT" use cases emerging whereby human actors circumvent the prescribed monitoring strategy in order to obtain the information necessary to their roles.

Second in order to identify the actors involved, it is necessary to identify the values, metrics, logs and other data that are collected as part of the monitoring strategy. This typically includes:

- Performance metrics e.g. cpu, queries/second, response time etc.
- Utilisation metrics, e.g. memory, bandwidth, disk, database tables etc.
- Throughput metrics e.g. network, caches, http etc.
- Log data from each host and application.
- User metrics including page views, click rates etc.
- Availability including uptime, host and service failure and network failure.
- Compliance data e.g. permissions, SLA related metrics, availability etc.
- Performance Indicators e.g. number of users, cost per transaction, revenue per hour.

Once these variables have been appropriately identified developing a monitoring strategy becomes a routing problem. One must devise the means to collect the aforementioned variables and deliver them to the appropriate actors in the appropriate format. Despite the claims made by many monitoring tools there is no single bullet which solves this problem. No single monitoring tool supports collection from all the required sources, provides all the necessary analysis or provides all the necessary outputs. Most monitoring strategies are therefore a patchwork of several monitoring tools which either operate independently or are interlinked by shared backends or dashboards.

Implementing a monitoring strategy

There exists a not insignificant number of monitoring tools, each designed for different, but sometimes overlapping, purposes. When faced with implementing a monitoring strategy there are three basic options with regards to software choices:

1. Use an existing monitoring tool.
2. Modify and existing monitoring tool to better fit the requirements specified by the strategy.
3. Implement a new bespoke tool which fits the requirements of the strategy.

There also exists a fourth option, which is to use a combination of the above options. For most general use cases, such as those involving a multi tiered web application built from standard software, a preexisting monitoring tool is likely to be sufficient to implement any monitoring strategy. Should the most appropriate tool lack features needed to fully implement the strategy, such as log parsing which is underrepresented in the feature lists of many monitoring tools, additional tools can be incorporated into the implementation of the strategy. Monitoring strategies which involve bespoke applications or simply uncommon applications can result in few tools supporting those applications. Most monitoring tools have some form of plugin architecture. This can range from Nagios which supports a menagerie of shell, perl and other scripts to more modern tools such as Riemann which has a modern functional API. These mechanisms allow monitoring tools to be extended to support data collection from bespoke or uncommon applications allowing tool choice not to be restricted by existing support for applications. Greater difficulties arise when necessary features are missing from a monitoring tool. Certain monitoring strategies can call for complex analysis, alerting, visualisation or other features which are unsupported by the vast majority of tools. Typically, such types of core functionality cannot be easily implemented via a plugin. This is not always the case but it is an issue which affects many tools. In which case there are two options: to add this functionality an existing tool or to design a new tool. The former requires knowledge of the original codebase and the ability to maintain a fork which may be prohibitive to smaller organisations. The latter option requires another tool be added to the strategy and it's coordination and communication be orchestrated. This latter option is what distinctly appears to be the most popular option. This can be evidenced by the number of open source tools which provide a small

part of a monitoring strategy that can be integrated as part of a larger strategy. Such tools include StatsD, Graphite, Graphene and HoardD which are all small tools providing a fragment of any monitoring strategy show the current trend towards implementing small bespoke tools as they are require in lieu of modifying larger more complex tools.

This trend is advantageous as it allows operations staff to choose from a diverse range of tools and devise an implementing which fits the monitoring strategy as closely as possible. This diversification of monitoring tools also represents many potential issues, namely that smaller and less popular tools may lose developers and support if it recedes from vogue. Furthermore a diverse range of tools requires a significant effort to orchestrate the communication between these tools. Unless there is a shared language, protocol or format the orchestration of these tools can become a troublesome.

Beyond software there is a human element to implementing monitoring strategies. There are numerous factors involved with implementing a monitoring strategy with regards to human actors, these including:

- Who is responsible for responding to alerts and events raised by the monitoring system and what oversight occurs when taking action.
- If an event occurs which the monitoring strategy fails to detect what action is taken to improve the strategy.
- Who is responsible for maintaining the monitoring system.
- How is access to the monitoring system controlled and how is access granted are all users of the system allowed to see the full data.

Even a monitoring system which is highly autonomic in nature still relies upon some degree of human interaction. Failure of human actors to take appropriate action is equally, if not more, problematic than software agents failing. It is therefore essential for human and software actors to abide by the monitoring strategy to ensure effective monitoring.

The chaos monkey and testing monitoring strategies

Few monitoring strategies are perfect and the worst time to discover imperfections in a monitoring strategy is in production when a service is under high demand. Chaos monkey is a term popularised by Netflix [98], which defines a user or piece of software which introduces failure or error during a pre approved period of time in order to test the effectiveness of a monitoring strategy and the response of the operations team. In the case of Netflix the chaos monkey terminates a random EC2 instance in order to simulate a common failure mode. This failure should be handled automatically by the monitoring strategy and accounted for without the need for human intervention. In the case that an unexpected outcome occurs the monitoring strategy is adapted to cover what was previously unexpected. The success of this testing strategy has led to Netflix open sourcing their Simian Army toolkit, a widely adopted tool which not only includes the Chaos Monkey but also several other programs which create latency, security, performance, configuration and other issues in order to test the effectiveness of a monitoring strategy.

This type of testing whether it is automated, manual or a mixture of both is essential in testing the effectiveness of a monitoring strategy. Failing to test a monitoring strategy is akin to failing to test software. Without appropriate testing, a developer cannot assert that their product meets the prescribed requirements or indeed make any strong empirical claims regarding the validity of their software. This is also true of a monitoring strategy. While software testing has been an intrinsic part of software engineering since it's beginnings, efficient testing methods for monitoring strategies are only beginning to emerge.

Changing knowledge requirements and job descriptions

In previous sections we have discussed the role of the operations team in a reasonably abstract manner. Operations is a core part of any large organisation and as technology, knowledge and work culture has evolved so to have the roles which comprise operations. For the last 20 years the maintenance of software and hardware systems has fallen under the preview of a system administrator. For the longest time operations and system administration were largely interchangeable. Sysadmins have typically been responsible for a broad and often ill-defined range of maintenance and deployment roles of which monitoring inevitably is a large component. System administrators are not developers or software engineers, they do not typically implement new software or modify current systems. Therefore when a monitoring tool reports a fault with an application it is beyond their responsibilities to delve into application code in order to resolve the issue and prevent its reoccurrence. Instead, sysadmins are typically limited to altering configuration, system settings and other tuneable parameters in order to resolve an issue. If a problem arises with an application which cannot be fixed by sysadmins then generally, the problem is communicated to development staff who then resolve the problem. Often communication between these two groups is facilitated through issue trackers, project management software or other collaboration tools. As monitoring is the prerogative of the administrator (and due to the administrator-developer divide) monitoring tools have typically produced performance metrics, status codes, failure notifications and other values that are of direct use to the administrator. This leads to administrators using monitoring tools in a reactive manner: they take corrective action when a monitoring tool alerts them.

This separation of responsibility between administration staff and development staff eventually became strained. With the ever increasing diversity and complexity of modern software it has become common place for organisations to deploy bespoke or heavily modified software. In such cases it is extremely beneficial for operations staff to have detailed knowledge of the inner workings of these systems. Furthermore, as monitoring tools have become more advanced it has become feasible to monitoring a vast range of application specific variables and values. It is therefore of importance that operations staff who make use of monitoring data have the prerequisite application knowledge to understand, interpret and take action upon monitoring data. Since the late 2000s what has, arguably, occurred has been a bifurcation of administration roles into administrators with broad knowledge of system architecture and application design and those without. The later is still typically referred to as a systems administrator while the later has assumed the moniker of DevOps.

DevOps rethinks the purpose of monitoring. Instead of the very reactive role monitoring has in conventional systems administration, DevOps takes a proactive stance to monitoring whereby monitoring becomes an intrinsic part of the development process. This leads to the promotion of a "monitor everything" approach whereby all potentially interesting variables are recorded as these values are now utilised throughout the development process, as opposed to simply the maintenance process. Conceptually, this improves the development process and ensures the design and implementation of software is grounded in empiricism.

While DevOps was gaining acceptance a separate trend has been emerging. Site Reliability Engineers (SRE) is a job role first created by Google which has since spread to other organisations. While DevOps attempts to merge the traditional system administrator and developer role, SREs are, for all intents and purposes, are software engineers tasked with an operations role. This new role has emerged to fill the needs of organisations for whom "infrastructure is code". When the division between all levels of infrastructure are blurred, as is the case now with cloud computing, operations roles require detailed knowledge of software engineering. With regards to monitoring, SRE take an equivalent "monitoring everything" approach and utilise detailed, high frequency monitoring data during all stages of the development and maintenance process.

The emergence of both DevOps and SRE is indicative of a trend away from non-programming system administrators and towards a far winder range of skills and knowledge. With this diversification of knowledge comes a change in who monitoring data is intended for. No longer are monitoring tools intended only for operations staff, instead they useful to stakeholders throughout engineering and management. Table 4 provides a list of monitoring stakeholders and their interests.

Monitoring as a data intensive problem

"Monitor Everything" is a phrase that features heavily in discourse regarding monitoring best practices. This certainly removes the risk that a monitoring strategy will fail to encompass all the relevant variables but if taken in the most literal sense can be extremely difficult to achieve. A VM instance has no shortage of data points, these includes: application metrics, network activity, file system changes, context switches, system calls, cache misses, IO latency, hypervisor behaviour and cpu steal to name but a few. The list is extensive. In the case of a private cloud whereby physical hosts are monitored there is an additional set of hardware values that can be monitored.

Collecting all of these variables is challenging, especially if a monitoring strategy calls for these variables to be monitored in real time. While the vast majority of these variables are small, the volume of variables and their rate of change risks all encompassing monitoring (or event more restricted monitoring) becoming a data intensive challenge. In the past, common practice has been sampling variables at 30 second or even greater increments. As an industry standard this is no longer acceptable. Most tools now operate at a 1 second interval. At scale, processing and storing these variables at this interval requires a significant volume of compute capacity.

Table 4 Monitoring stakeholders

Stakeholder	Motivation
System Administrator	Detecting failure, ensuring continued operation of system
DevOps	Data led software development, system maintenance
Security Staff	Intrusion detection, enforcement and protection
SRE	Software Engineering orientated operations
Customers	Billing
Testers	Input for regression, requirements and performance testing
Product Owner	Ensure product quality and ensure business value
Scrum Master	Ensure adherence to SCRUM
Developers	Debugging, identify performance issues, trace messages and transactions
UI Team	Observe user behaviour
Business Stakeholders	Predict future requirements, costs and profitability
Compliance Auditors	SLA compliance, certification auditing

The trend towards all encompassing monitoring is reflected in the monitoring tools. As our survey demonstrates, older monitoring tools utilised predominantly centralised architectures whereby a single server collected data from monitored hosts. With larger systems or with an all encompassing monitoring strategy, this scheme is no longer viable. Newer tools offer schemes where the computation involved in monitoring can be distributed over a large number of hosts using message queues, middleware, shared memory or other communication channels.

This complicate the problem of monitoring vasty. No longer do operations staff have to contend with the management of a single monitoring server but rather with a a monitoring cluster which is likely proportional in size to the monitored deployment. This had led to the development of many cloud hosted service whereby this management complexity is abstracted behind an web service, freeing operations staff from managing large monitoring systems. This option is however, unacceptable for many organisations who due to compliance, legal, security or other reasons require that their monitoring system be behind their firewall. This creates a number of issues. Operations staff at organisations with large systems must now contend with maintaining large monitoring systems in addition to their production systems. These issues are also concerning for organisations which do not regularly operate at scale but by using cloud computing to autoscale their infrastructure can temporarily operate at a larger scale which requires larger and more costly monitoring.

Conclusion

Cloud computing is a technology which underpins much of today's Internet. Central to its success is elasticity; the means to rapidly change in scale and composition. Elasticity affords users the means to deploy large scale systems and systems which adapt to changes in demand. Elasticity also demands comprehensive monitoring. With no physical infrastructure and a propensity for scale and change it is critical that stakeholders employ a monitoring strategy which allows for the detection of problems, optimisation, cost forecasting, intrusion detection, auditing and other use cases. The lack of an appropriate strategy risks downtime, data loss, unpredicted costs and other unwanted outcomes. Central to designing and implementing a monitoring strategy are monitoring tools. This paper has exhaustively detailed the wide range of monitoring tools related to cloud monitoring. As is evident, relevant tools originate from a number of domains and employ a variety of designs. There are a considerable number of venerable tools originating from enterprise, grid and cluster computing which have a variety of appropriateness to cloud monitoring. More recent developments have seen the development of cloud specific monitoring tools. These tools are either installed by users on infrastructure or operate as software as a service tools. At present there is no universally accepted means to compose a monitoring strategy or to select the appropriate tools. At present most monitoring strategies are a patchwork of several monitoring tools each which provide different functionality. Common schemes include several tools which between them provide metric gathering, log/event processing and health and network monitoring.

Many older tools which originate from previous domains are poor fits for cloud computing as they rely on centralised strategies, pull models, manual configuration and other anachronisms. Newer tools attempt to avoid these issues but as yet do not provide the full gamut of functionality offered by existing tools. Our taxonomy demonstrates that while newer tools better support the requirements of cloud monitoring they do not yet have the entire range of necessary functionality. A trend which is clear from our survey is the gradual transition from grid,cluster and enterprise monitoring tools to more appropriate cloud monitoring tools. This trend will increase greatly as more modern tools develop the range of plugins, software support and functionality that older tools currently support.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

JSW conducted the detailed survey of monitoring frameworks as part of his PhD studies. AB helped develop the structure, analysis and taxonomy for the survey and provided substantial input into drafting and revising the document. Both authors read and approved the final manuscript.

Acknowledgements

This research was supported by a Royal Society Industry Fellowship and an Amazon Web Services (AWS) grant.

Received: 6 October 2014 Accepted: 10 December 2014
Published online: 29 December 2014

References

1. Tremante M (2013) Amazon Web Services' growth unrelenting. http://news.netcraft.com/archives/2013/05/20/amazon-web-services-growth-unrelenting.html
2. Mendoza D (2012) Amazon outage takes down Reddit, Foursquare, others - CNN.com. http://edition.cnn.com/2012/10/22/tech/web/reddit-goes-down/, 2012
3. Mell P, Grance T (2011) The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD
4. Armbrust M, Stoica I, Zaharia M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A (2010) A view of cloud computing. Commun ACM 53(4):50
5. Subramaniyan R, Raman P, George AD, Radlinski M (2006) GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems. Cluster Computing 9(1):101–120
6. Povedano-Molina J, Lopez-Vega JM, Lopez-Soler JM, Corradi A, Foschini L (2013) DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds. Future Generation Comput Syst 29(8):2041–2056
7. Konig B, Alcaraz Calero JM, Kirschnick J (1306) Elastic monitoring framework for cloud infrastructures. IET Commun 6(10)
8. Van Renesse R, Birman KP, Vogels W (2003) Astrolabe. ACM Trans Comput Syst 21(2):164–206

9. Riemann. http://riemann.io/
10. cloudinit.d. http://www.nimbusproject.org/doc/cloudinitd/1.2/
11. El-Khamra Y, Kim H, Jha S, Parashar M (2010) Exploring the Performance Fluctuations of HPC Workloads on Clouds. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science. IEEE. pp 383–387
12. Amazon Web Services FAQs. https://aws.amazon.com/ec2/faqs/
13. Machine Types - Google Compute Engine - Google Developers. https://developers.google.com/compute/docs/machine-types
14. Got Steal? | CopperEgg. http://copperegg.com/got-steal/
15. Link D (2011) Netflix and Stolen Time. http://blog.sciencelogic.com/netflix-steals-time-in-the-cloud-and-from-users/03/2011
16. Avresky DR, Diaz M, Bode A, Ciciani B, Dekel E, eds (2010) A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing, volume 34 of Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg
17. Top500.org Amazon EC2 Cluster Compute Instances - Amazon EC2 Cluster, Xeon X5570 2.95 Ghz, 10G Ethernet | TOP500 Supercomputer Sites. http://www.top500.org/system/10661
18. BuisnessWeek Another Amazon Outage Exposes the Cloud's Dark Lining - Businessweek. http://www.businessweek.com/articles/2013-08-26/another-amazon-outage-exposes-the-clouds-dark-lining
19. ZDNet Amazon Web Services suffers outage, takes down Vine, Instagram, others with it, ZDNet. http://www.zdnet.com/amazon-web-services-suffers-outage-takes-down-vine-instagram-flipboard-with-it-7000019842/
20. Ec2 Global Infrastructure. https://aws.amazon.com/about-aws/globalinfrastructure/?nc1=h_l2_cc
21. Azure Microsoft Azure Trust Center - Privacy. http://azure.microsoft.com/en-us/support/trust-center/privacy/
22. Ward JS, Barker A (2012) Semantic based data collection for large scale cloud systems. In: Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date. ACM, New York, NY, USA. pp 13–22
23. Foster I, Zhao Y, Raicu I, Lu S (2008) Cloud computing and grid computing 360-degree compared. In: Grid Computing Environments Workshop, 2008. GCE'08. IEEE. pp 1–10
24. Aceto G, Botta A (2013) Walter de Donato and Antonio Pescapè. Cloud monitoring: A survey. Comput Netw 57(9):2093–2115
25. Deelman E, Singh G, Livny M, Berriman B, Good J (2008) The cost of doing science on the cloud: The Montage example.1–12
26. Greenberg A, Hamilton J, Maltz DA, Patel P (2008) The cost of a cloud. ACM SIGCOMM Comput Commun Rev 39(1):68
27. Sarathy V, Narayan P, Mikkilineni R (2010) Next Generation Cloud Computing Architecture: Enabling Real-Time Dynamism for Shared Distributed Physical Infrastructure. In: 2010 19th IEEE International, Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises. IEEE. pp 48–53
28. Ian Foster, Yong Zhao, Ioan Raicu, Shiyong L u Cloud Computing and Grid Computing 360-Degree Compared. In: 2008 Grid Computing Environments Workshop. IEEE. pp 1–10
29. Kozuch MA, Ganger GR, Ryan MP, Gass R, Schlosser SW, O'Hallaron D, Cipar J, Krevat E, López J, Stroucken M (2009) Tashi. In: Proceedings of the 1st workshop on, Automated control for datacenters and clouds - ACDC '09. ACM Press, New York, New York, USA. p 43
30. Marshall P, Keahey K, Freeman T (2011) Improving Utilization of Infrastructure Clouds. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE. pp 205–214
31. Massie ML, Chun BN, Culler DE (2004) The ganglia distributed monitoring system: design, implementation, and experience. Parallel Comput 30(7):817–840
32. Tobias Oetiker RRDtool. http://oss.oetiker.ch/rrdtool/
33. Eisler M XDR: External Data Representation Standard. http://tools.ietf.org/html/rfc4506
34. Daniel Pocock ganglia-nagios-bridge. https://github.com/ganglia/ganglia-nagios-bridge
35. Nagios Nagios - The Industry Standard in IT Infrastructure Monitoring. http://www.nagios.org/
36. Intellectual Reserve Distributed Nagios Executor (DNX). http://dnx.sourceforge.net/
37. NagiosConfig Main Configuration File Options. http://nagios.sourceforge.net/docs/3_0/configmain.html
38. Kowall J Got Nagios? Get rid of it. http://blogs.gartner.com/jonah-kowall/2013/02/22/got-nagios-get-rid-of-it/
39. Collectd collectd – The system statistics collection daemon. http://collectd.org/
40. Rightscale Cloud Portfolio Management by RightScale. http://www.rightscale.com/
41. Logstash. http://logstash.net/
42. statsd. https://github.com/etsy/statsd/
43. Bucky. https://github.com/trbs/bucky
44. Graphite. http://graphite.wikidot.com/
45. drraw. http://web.taranis.org/drraw/
46. Cabot. http://cabotapp.com/
47. Google (2014) Google Protocol Buffers. https://developers.google.com/protocol-buffers/docs/overview
48. sFlow. http://www.sflow.org/
49. Host sFlow. http://host-sflow.sourceforge.net/
50. PeakFlow. http://www.arbornetworks.com/products/peakflow
51. Scrutinizer sFlow Analyzer. http://www.plixer.com/Scrutinizer-Netflow-Sflow/scrutinizer.html
52. MonALISA. http://monalisa.caltech.edu/monalisa.htm
53. Legrand I, Newman H, Voicu R, Cirstoiu C, Grigoras C, Dobre C, Muraru A, Costan A, Dediu M, Stratan C (2009) MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems. Comput Phys Commun 180(12):2472–2498
54. Arnold K, Scheifler R, Waldo J, O'Sullivan B, Wollrath A (1999) Jini Specification. http://dl.acm.org/citation.cfm?id=554054
55. Lee D, Dongarra JJ, Ramakrishna RS (2003) visperf: Monitoring tool for grid computing. In: Sloot PMA, Abramson D, Bogdanov AV, Gorbachev YE, Dongarra JJ, Zomaya AY (eds) Computational Science – ICCS 2003 volume 2659 of Lecture Notes in Computer Science. Springer Berlin Heidelberg. pp 233–243
56. Kermarrec A-M, Massoulie L, Ganesh AJ (2003) Probabilistic reliable dissemination in large-scale systems. IEEE Trans Parallel Distributed Syst 14(3):248–258
57. GEMS Java Implementation. https://github.com/nsivabalan/gems
58. OmniTI Labs (2014) Reconnoiter. https://labs.omniti.com/labs/reconnoiter
59. Icinga. https://www.icinga.org/
60. Zenoss (2014) Zenoss. http://www.zenoss.com/
61. The Cacti Group (2014) Cacti. http://www.cacti.net/
62. GroundWork. http://www.gwos.com/
63. Munin. http://munin-monitoring.org/
64. OpenNMS The OpenNMS Project. http://www.opennms.org/
65. Spiceworks Spiceworks. http://www.spiceworks.com/
66. Zabbix Zabbix. http://www.zabbix.com/
67. cloudinit.d monitoring README. https://github.com/nimbusproject/cloudinit.d/blob/master/docs/monitor.txt
68. Sensu. http://sensuapp.org/
69. RabbitMQ. https://www.rabbitmq.com/
70. Bloom A How fast is a Rabbit? Basic RabbitMQ Performance Benchmarks. https://blogs.vmware.com/vfabric/2013/04/how-fast-is-a-rabbit-basic-rabbitmq-performance-benchmarks.html
71. An K, Pradhan S, Caglar F, Gokhale A (2012) A Publish/Subscribe Middleware for Dependable and Real-time Resource Monitoring in the Cloud. In: Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management. ACM, New York, NY, USA. pp 3:1–3:6
72. Goldsack P, Guijarro J, Loughran S, Coles A, Farrell A, Lain A, Murray P, Toft P (2009) The SmartFrog configuration management framework. ACM SIGOPS Operating Syst Rev 43(1):16
73. Dhingra M, Lakshmi J, Nandy SK (2012) Resource Usage Monitoring in Clouds. In: Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing. IEEE Computer Society, Washington, DC, USA. pp 184–191
74. Pardo-Castellote G (2003) OMG data-distribution service: architectural overview. In: 23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings. IEEE. pp 200–206
75. Kung HT, Lin C-k, Vlah D (2011) CloudSense : Continuous Fine-Grain Cloud Monitoring With Compressive Sensing.

76. Candes EJ, Wakin MB (2008) An Introduction To Compressive Sampling. Signal Processing Magazine, IEEE:21–30
77. Montes J, Sánchez A, Memishi B, Pérez MS, Antoniu G (2013) GMonE: A complete approach to cloud monitoring. Future Generation Comput Syst 29(8):2026–2040
78. Lakshman A, Malik P (2010) Cassandra: A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. 44(2):35–40
79. Clayman S, Galis A, Mamatas L (2010) Monitoring virtual networks with Lattice. In: Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP. pp 239–246
80. OpenNebula. http://opennebula.org/
81. OpenNebula KVM and Xen UDP-push Monitoring. http://docs.opennebula.org/4.4/administration/monitoring/imudppushg.html
82. Melis J OpenNebula 4.4: New Massively Scalable Monitoring Driver. http://opennebula.org/opennebula-4-4-new-massively-scalable-monitoring-driver/
83. De Chaves SA, Uriarte RB, Westphall CB (2011) Toward an architecture for monitoring private clouds. Communications Magazine, IEEE 49(12):130–137
84. PCMONS Download. https://github.com/pedrovitti/pcmons
85. Eucalyptus: Open Source Private Cloud Software. https://www.eucalyptus.com/eucalyptus-cloud/iaas
86. Ward JS, Barker A (2013) Varanus: In Situ Monitoring for Large Scale Cloud Systems. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science. IEEE Vol. 2. pp 341–344
87. Amazon CloudWatch. http://aws.amazon.com/cloudwatch/
88. HP Cloud Monitoring. http://www.hpcloud.com/products-services/monitoring
89. HP Public Cloud. http://www.hpcloud.com/
90. Monitoring System - RightScale. http://support.rightscale.com/12-Guides/RightScale_101/08-Management_Tools/Monitoring_System
91. New Relic. http://newrelic.com/
92. CopperEgg. http://copperegg.com/
93. CloudSleuth. https://cloudsleuth.net/
94. Network & IT Systems Monitoring | Monitis - Monitor Everything. http://www.monitis.com/
95. Cloud Monitoring as a Service for AWS and Rackspace. https://www.stackdriver.com/
96. Boundary: Unified Monitoring for Web-Scale IT and Modern IT Operations Monitoring. http://boundary.com/
97. Multi cloud Optimization: AWS & Google Cloud Services | Cloud Analytics | Cloudyn | Actionable Recommendations. http://www.cloudyn.com/
98. Netflix Chaos Monkey. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html