Tablespoon - real-time system metric monitoring for Karamel

SEBASTIAN TÖRNQVIST
HENRIK WESTERBERG
Tablespoon - real-time system metric monitoring for Karamel

Westerberg, Henrik
hwesterb@kth.se

Törnqvist, Sebastian
stornq@kth.se

1st September 2016
Abstract

System resource utilisation metrics is an important source of the decision-making process for a general-purpose auto-scaling solution in cloud computing. It is critical for a monitoring system to be light-weight in its usage of system resources. In this work, Tablespoon, a real-time monitoring system, is presented. It operates on a publish-subscribe architecture and is agent push-based. Tablespoon itself has a low bandwidth usage profile by using agent-side filtering and an inter-group aggregation mechanism. Our solution ensures that requested events are received at most once by the subscriber, whilst simultaneously minimising the loss of such events. The evaluation of Tablespoon shows that the average latency, in a limited use case, between publisher and subscriber, is around 500-700ms. Furthermore, the average CPU usage of the agents, depending on the use case, is about 0.5 percent when testing on Amazon Web Services.
Acknowledgements

We would like to express our sincere gratitude to our supervisor Kamal Hakimzadeh for his invaluable patience and guidance throughout the thesis. In addition, we wish to thank our examiner Jim Dowling for sharing his knowledge and experience in the area. At last, we are thankful to all employees at the SCS department that we have spoken to during the thesis, giving us their advice and support.
Contents

List of Figures 6
List of Tables 6

1 Introduction 7
1.1 Problem ............................... 7
1.2 Purpose ................................ 8
1.3 Goals ................................ 8
1.4 Benefits, ethics and sustainability ......... 9
1.5 Methodology ............................ 9
1.6 Delimitations ........................... 9
1.7 Outline ................................ 9

2 Background 10
2.1 Environment of Tablespoon ............... 10
2.2 Cloud Computing ........................ 10
  2.2.1 Definition ........................... 11
  2.2.2 Payment model ........................ 11
  2.2.3 Virtualization ........................ 11
  2.2.4 Obstacles and limitations .............. 12
  2.2.5 Comparing cloud providers ............. 12
2.3 Cloud Monitoring ........................ 13
  2.3.1 Motivations for cloud monitoring ....... 13
  2.3.2 Attributes of a cloud monitoring system . 14
2.4 Cloud Orchestration ....................... 16
  2.4.1 Chef ................................ 16
  2.4.2 Karamel ............................. 16
2.5 Event Stream Processing .................... 16
  2.5.1 Riemann .............................. 17
2.6 Elastic Scaling ........................... 17
2.7 Concurrency ............................. 18
  2.7.1 Concurrency in Java ................... 18
2.8 Related Work ............................ 19
  2.8.1 Collectl monitoring ................... 19
  2.8.2 Cloudify stack ....................... 20
  2.8.3 Riemann-InfluxDB-Grafana stack ....... 20

3 Methods and Methodologies 21
3.1 Research Methodology ..................... 21
3.2 Code Testing ............................ 22
  3.2.1 Unit tests ........................... 22
  3.2.2 Integration tests ..................... 22
3.3 System Evaluation Method ................. 22

4 Tablespoon in Karamel 23
4.1 Tablespoon ............................. 23
4.2 Agent ................................ 24
4.3 Server ................................ 25
List of Figures
1 Overview of Tablespoon architecture........................... 23
2 Flow chart of agent ............................................. 24
3 Agent sends batch to Riemann server.......................... 25
4 Pseudocode of Tablespoon’s Riemann configuration......... 26
5 Example showing the construction of API calls............... 27
6 A state diagram depicting the states of AgentBroadcasterAssist-
ant................................................................. 28
7 Comparison between different user scenarios .................. 33
8 Graphs showing the results of the latency experiment ....... 34

List of Tables
1 Twelve characteristics of a monitoring system (as described in [1]). 15
2 Table explaining features of the API ............................ 28
3 Showing evaluation results of cloud monitoring characteristics. 34
1 Introduction

Cloud computing is rapidly growing and today there are several significant actors on the market, such as Amazon EC2, Microsoft Azure and Google Cloud Platform [2, 3, 4]. In a recent study it was shown that the most alarming issue for customers is the lack of expertise and resources [5]. Karamel is an easy-to-use cloud management application that targets this issue. It is an orchestration tool for large scale clusters built with the purpose of making cloud computing more accessible to the general audience. It offers the ability to run reproducible experiments, which are sequences of commands executed distributedly. However, in order to achieve dynamic scaling and optimisation of cloud resource usage, Karamel needs system metrics from the cloud instances. The current monitoring system in Karamel is limited to visualising the system utilisation of an experiment subsequently and is therefore not able to meet the needs of auto-scaling. This thesis presents Tablespoon which monitors system metrics in real-time, thus enabling reactive decision-making and improving the monitoring capability within Karamel.

1.1 Problem

Traditional data centres are fixed in its ability to provide storage and computing resources. Cloud computing’s dynamic provisioning of resources leads to a decrease in revenue loss, while still being able to cater a fluctuating load of users. There are several ways of providing this adaptability; from manually buying new resources, to using tools administered by cloud providers or third-party alternatives. To make an informed choice monitoring data is always needed.

The problem area of this thesis lies within providing system-level monitoring metrics. This data must be collected, processed and then delivered to its respective subscriber. As the clusters grow and change, the monitoring tool must adapt to the new conditions. Many components, from event-processing servers, to agents collecting data, to publishers and subscribers, must be able to receive information in a reliable way. Subscribers may have different needs, GUI-components may only need a summary of performance load while others, like load balancing components, may need a rapid stream of events in real-time to make reliable decisions.

A survey on cloud monitoring mentions twelve advantageous characteristics of a cloud monitoring system. For every property, they outline the main research issues that come with meeting the criteria. For scalability the main issues are aggregation and filtering of measurements, for timeliness it is processing large amounts of data and for autonomicity it is situation awareness [1].

When designing a large scale system that runs on a cloud infrastructure, architectural decisions have a significant impact on what becomes the strengths and weaknesses of the created system. When making these decisions it is important to take into account for what reason and by whom the system is to be used. Is the data to be consumed right away or does it need to be stored? Is some loss of data tolerable? Is multi-tenancy a desired feature? How important is security?
Tablespoon is designed to meet the needs of the Elastic Scaler in Karamel, where timeliness is crucial since decisions are made continuously in real-time. The type of application that a user chooses to run together with Karamel is difficult to anticipate, thus emphasising the demand for configurability by the user. Therefore, the problem of this thesis is summarised into following question: How to design a monitoring system that is lightweight in its usage of resources, but also timely and configurable?

1.2 Purpose

The purpose of this thesis is to describe and discuss the progress and result of the development of Tablespoon. Tablespoon is an artefact developed iteratively with the design science research approach. The tool is tested with the experimental research method. The work presents design choices for creating a multi-cloud monitoring architecture that focuses on minimal impact on network and machine resources.

The purpose of Tablespoon is to aid Karamel’s Elastic Scaler in making decisions about resource provisioning. An additional purpose is to offer the user an overview of system resources. This view enhances the understanding of the Chef recipes and tasks that are executed by Karamel and of the distributed algorithms that run on the multi-cloud infrastructure.

1.3 Goals

The goal of this thesis is to provide a monitoring system tailored for Karamel that does not severely impact the performance. Furthermore, it should have the ability to be reconfigured whenever the needs of the Karamel system changes.

One of Karamel’s core design principles is being easily accessible for users, Tablespoon should also conform to this idea, by using intelligible graphical and programming interfaces. The entirety of the monitoring system should not notably affect the performance of the virtual machine instances, the network or the local machine. To achieve this Tablespoon should implement a lightweight network communication protocol to ensure messages sent are compact and not using extensive network bandwidth. Moreover, the agents collecting the metrics should not obstruct the application that is run on the node. It should also retain metrics in a reliable way to avoid any memory leakage. Filtering is another important aspect. It could be done by the agent, the event stream processor or on the local machine. A goal is therefore to find the best practises for filtering as well as understanding the trade-offs that come from choosing one method over another.

Employing good software principles is part of the scope of this thesis. In design science research, when developing an IT-artefact, the artefact is evaluated in terms of utility, quality, and efficacy. Code quality directly impacts that evaluation. Short, reusable, generic units of code and loosely coupled, proportionate and distinct components facilitate the maintainability of the code base. Tablespoon is an open source project which relies on version control, build
automation and unit testing. Those technologies should, therefore, be used in a consistent and correct manner.

1.4 Benefits, ethics and sustainability

The Tablespoon system will benefit users of large scale distributed clusters who desire open-source multi-cluster monitoring instead of the variants coupled with the cluster distributor, such as for example Amazon CloudWatch [9]. From a sustainability perspective Tablespoon would result in a more effective cloud resource management meaning, for example, a reduction in power usage. During 2013 and 2014, a spiked interest for the energy consumption dimension of cloud computing wa observed within the scientific community [10].

1.5 Methodology

The thesis will evaluate the system developed in a qualitative fashion. By applying the design science research method this thesis will show that the goals have been reached. This is demonstrated by qualitatively evaluating the system developed for given parameters and showing that each of them is fulfilled.

1.6 Delimitations

This report does not discuss all other related systems that are applicable to the problem. No extensive literature study was conducted about the current state of the art within cloud monitoring systems. Instead this thesis relies the 2013 paper "Cloud Monitoring: A survey". For this reason some new developments within cloud monitoring might have gone unnoticed.

Furthermore, if the degree project were to include a graphical user interface as was first planned, some of the system architectural decisions might have been different, by for instance changing the method of receiving events to better suit the GUI.

1.7 Outline

This thesis presents and evaluates the monitoring system Tablespoon and its integration into Karamel. Chapter 2 begins by giving a general background to the area of monitoring in distributed systems and continues by introducing other related systems. In Chapter 3 the methodologies for validating the work are presented. Chapter 4 displays the Tablespoon system. The evaluation of the system is presented in Chapter 5. Moreover, the evaluation results are discussed in Chapter 6. Chapter 7 concludes the thesis and also suggest future work.
2 Background

This chapter gives a description of the concepts that are relevant to the thesis. It then continues by describing the specific technologies related to Tablespoon.

2.1 Environment of Tablespoon

Tablespoon works as a module part of the Karamel ecosystem. Much of its functionality, for example orchestration, node management and failure detection is brought by Karamel. Consumers of the metrics provided by Tablespoon are either other modules inside of Karamel or users of the Karamel system. For this reason a brief introduction to Karamel and Cloud Orchestration is given in Section 2.4 on page 16.

Honeytap\[11\] is an Elastic Scaler developed in conjunction with Tablespoon. Tablespoon has been tailored to meet the needs of this component. The concept of elastic scaling is described in Section 2.6 on page 17.

Tablespoon uses an Event Stream Processor to aggregate large volumes of information sent by agents. To send and query for events Tablespoon uses the Riemann Client. Riemann is presented in Section 2.5.1 on page 17.

The language used throughout Tablespoon and Karamel is Java. Since Tablespoon is real-time and capable of serving several subscribers, a lot of focus was put on ensuring concurrency. How concurrency is implemented in Java is presented in Section 2.7 on page 18.

2.2 Cloud Computing

Many modern computer applications are either in or moving into, the realm of big data. In such systems computing power and data needs to be distributed over complex network architectures. The main sources of big data are operation and trading information in enterprises, logistic and sensing information in Internet of Things (IoT), information describing human interaction with systems and data generated for scientific research. This amount far surpasses what is currently available in data centres. \[12\]

As this need grows many organisations find it more beneficial to obtain this service from outside providers. Cloud computing is a term to describe these services. The technology is gaining popularity within the scientific community. Bayramusta and Nasir analysed 236 scholarly journal articles between 2009-2014 about cloud computing, from two research databases: ABI/INFORM Complete, Emerald and found a significant increase in published articles. A majority of the articles concerned how to adopt cloud computing to current business solutions. \[10\]
2.2.1 Definition

According to the National Institute of Standards and Technology (NIST) definition [13] the resources of a cloud can be rapidly provisioned and released with minimal human interaction. They are made available with network access and can, therefore, be consumed by various thin or thick client platforms. The provider utilises a multi-tenant model with physical and virtual resources which can be dynamically assigned. Providers optimise and control resources by monitoring suitable parameters.

NIST identifies three main cloud computing service models. Infrastructure as a Service (IaaS), which grants consumers control over operating systems, storage, and deployed applications. Platform as a Service (PaaS), which lets consumers install and maintain applications, access an API defined by the provider but restricts access to the operating system. Lastly, there is Software as a Service (SaaS), which provides some application maintained and developed by the provider. Amazon Elastic Compute Cloud (EC2) is a well-known IaaS service, Microsoft Azure and Google App Engine are PaaS services and Gmail, Google Docs, Salesforce.com are examples of SaaS services [10, pp.637].

2.2.2 Payment model

A consumer has to rely on cloud providers to maintain a certain Quality of Service (QoS). The requirements are enforced by service level agreements (SLA). For that reason providers needs to focus on market-oriented resource management, where resources are distributed to fulfil SLAs most efficiently.

In recent years, cloud providers are moving towards offering different pricing models. On-demand is a model where the customer is billed subsequently based on resources used. In the reservation model the user pays for the machine immediately and then secures availability. Lastly, there is the spot instance which can be considered the most flexible way of renting resources. It is derived from an auction-like model where the price for a machine is based on demand and supply. The spot instances can be terminated at any time if the market price exceeds the bidding price. [14]

2.2.3 Virtualization

Virtual machines (VM) are emulations of operating systems from which applications can be run. Virtualization allows for higher flexibility and the ability for cloud providers to maximise system resource utilisation. Several of these instances can run in parallel on the same physical machine and each instance is isolated from the underlying hardware. [15] Virtualization techniques allows cloud computing to transfer virtual machine instances from one server to another (as cited in [10, pp.637]).

A virtual machine monitor (VMM) is used to spawn VM instances in the cloud. Providers offer options for consumers to manage this interaction via Internet-based consoles. However, research is showing that a current trend among developers is to use more than one cloud provider. Hence, there is a need for
standardised API interaction. Graham and Liu evaluated two Open APIs solutions, JClouds and Cloudify. Both supports elastic scaling, however, Cloudify has built-in metrics gathering while JClouds has the ability to listen for metrics if exposed by the provider, a feature only supported by EC2 in this particular study. [16]

2.2.4 Obstacles and limitations

In distributed systems bottlenecks are prevalent. Parallel processing algorithms utilise the intra-cloud network heavily, if individual servers are using slower connection speeds, such as 1 Gigabit Ethernet (1GbE), it might become a bottleneck for the overall processing speed. The fact that a cloud uses virtualized resources introduces some performance unpredictability, such as I/O interference which affects read and write speeds. [17, pp.14-19] In recent years, some processing algorithms, like for instance Apache Spark, utilises the memory for storing data while performing a query.

The CAP theorem describes some well-known limitations with distributed systems. A distributed system can only ensure two of the three properties: availability (A), consistency (C) and network partitioning (P). Thus, there are CA, CP and AP systems. CA systems can be described as small-scale relational databases. Therefore, large scale distributed systems tend to be focused on either delivering consistency or availability. The AP system, focused on bringing availability, is best suited for situations where inconsistent data is tolerable, such as in Social Networking Services (SNS). While CP systems are preferable in batch processing that strives for high accuracy. [12, pp.185-186]

2.2.5 Comparing cloud providers

Cloud providers have been observed to have diverse characteristics. Yang et al. [18] describes measurements done in order to determine differences in regards to computational, storage and networking capabilities. The study conducted in 2010 involves four providers: AWS, Azure, Google App Engine, and Cloud-Servers. When presenting the results the companies names are replaced with an alias.

Three of the providers offer different machine specifications at varying prices. Higher-end configurations oftentimes include more cores. Other, not disclosed to the consumer, attributes like the amount of resource contention or CPU specification might also differ. To measure this difference they run single and multi-core benchmarks on each virtual instance specification. As a metric they use the time it takes for the benchmark to complete. They also take into account the cost of renting that particular virtual instance and from that they develop a price-performance ratio.

The results vary a lot with one provider being notably cheaper than the others. In regards to I/O performance, they noticed that multi-threaded configurations had worse performance. They theorise that this was due to it being harder to optimise interleaved requests from multiple threads. To conclude networking
capabilities they measure both within the cloud and within the wide area networks (WAN) which the cloud operates in. One of the providers only has 200 Mbps intra-cloud TCP throughput while the others have four times the amount.

They measure scaling latency by looking at the time it takes for a provider to allocate and make available new resources. Specifically, the time it takes to boot together with the time it takes to provision. Booting latency can be attributed to the type of operating system and the machine’s CPU and hard drive capacity. Other factors might include background services like monitoring or automatic fault-recovery that has to be prepared. The total latency was about ten times higher on some occasions.

In conclusion, providers excel at different aspects of cloud computing. Their cost models might not be completely fair and accurate in every regard. This data, however, is only a snapshot in time and the authors, therefore, conclude that continuous measurements would be beneficial.

2.3 Cloud Monitoring

Monitoring a cloud is imperative for quantifying the performance required by applications. Virtualized resources might move from one machine to another at any time, which implies new network and performance conditions [1, pp. 2095]. New conditions may also arise from network partitioning as mentioned in Section 2.2.4 on the preceding page or an increase in network traffic affecting that node. Without a cloud monitoring system it is hard to react to and troubleshoot such changes.

2.3.1 Motivations for cloud monitoring

Both providers and consumers are benefited by monitoring. Monitoring is mandatory when ascertaining SLAs, providers must be able to prove the quality of their service. When managing a complex and dynamic set of data centres, reliable and timely metrics is needed to make cost-effective decisions. If some part of the system is malfunctioning providers must be able to find the root cause, which can be very difficult with several layers of architecture like network links and physical and virtual instances. [1, pp.2095-2097]

For a consumer, most of the cloud is essentially a black box with a hidden internal infrastructure managed by another company. In the study mentioned in Section 2.2.5 on the previous page providers show a varying ability at bringing network, CPU and I/O performance. The price-performance ratio also differed greatly depending on which configuration the customer chose. Choosing the right configuration, at the most benign location, involves analysing measurements. As a monitoring system allows for a continuous stream of these kinds of measurements, such a decision can be based on the long-term performance of a provider.

The consumer might be running mission critical applications that require a certain throughput. Scientific applications must be proven reproducible and may
not tolerate variability. In any case knowledge about the underlying conditions is beneficial. Services that manage confidential records, for instances health care services, require Quality of Protection (QoP), which monitoring plays a role in providing. [1, pp.2095-2097]

In addition to the system associated metrics, there are metrics that are related to an application. For instance, the amount of users served, the average time to serve one user or notification of error messages. These kind of metrics are important for many applications.

2.3.2 Attributes of a cloud monitoring system

The aforementioned twelve advantageous characteristics are summarised in Table 1 on the following page.
<table>
<thead>
<tr>
<th>Monitoring Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>A scalable monitoring system can cope with a high and fluctuating number of probes without overburdening itself. Often attained by putting a relay on agents to limit sent data, aggregation and filtering and introducing sub-systems for coping with the influx of messages.</td>
</tr>
<tr>
<td>Elasticity</td>
<td>An elastic monitoring system can cope with resources being created and destroyed. Many traditional monitoring systems were not designed for this. Can be attained by using a VMM to keep track of live machines, push notifications from agents or using brokers to discover agents.</td>
</tr>
<tr>
<td>Adaptability</td>
<td>An adaptable monitoring system aims to be noninvasive. When computational and network requirements within the cloud change, those new requirements should not be obstructed by the monitoring system. In such situations, the monitoring system should adapt itself by decreasing the amount of metrics that are being collected and the frequency of sending those metrics.</td>
</tr>
<tr>
<td>Timeliness</td>
<td>A timely monitoring system provides metrics that are available at the time of their intended use. The duration of delivering a metric can be broken down into sampling, analysis and communication delay.</td>
</tr>
<tr>
<td>Autonomicity</td>
<td>An autonomic monitoring system can take the appropriate action to changes, faults and performance degradation on distributed resources without providers or consumers having to intervene.</td>
</tr>
<tr>
<td>Comprehensive-ness</td>
<td>A comprehensive monitoring system support several types of monitoring data and multiple tenants. If comprehensive enough eliminates the need for more than one monitoring API.</td>
</tr>
<tr>
<td>Extensibility</td>
<td>An extensible monitoring system can have its functionality extended by plug-ins or modules.</td>
</tr>
<tr>
<td>Low intrusiveness</td>
<td>A monitoring system with low intrusiveness is, in its implementation, isolated from and has low a impact on the normal operations of the cloud.</td>
</tr>
<tr>
<td>Resilience</td>
<td>A resilient monitoring system can withstand several components failing. In other words, no single point of failure (SPOF) threats on any of the components.</td>
</tr>
<tr>
<td>Reliability</td>
<td>A reliable monitoring system performs as stated under a defined set of conditions. It possible to predict its behaviour during a specified time frame.</td>
</tr>
<tr>
<td>Availability</td>
<td>An available monitoring system is accessible when it is requested.</td>
</tr>
<tr>
<td>Accuracy</td>
<td>An accurate monitoring system collects and processes measurements as close to the actual values as possible. Having shared resources like CPUs, interface cards, and buffers in virtual instances may lead to inaccuracy. Another problem is in regards to workload; when the system is heavily stressed some measurements, like for instance timestamps, might become imprecise.</td>
</tr>
</tbody>
</table>
2.4 Cloud Orchestration

Managing application running on IaaS clusters can be challenging and cloud orchestration tools have therefore been emerging.

2.4.1 Chef

Chef is a configuration framework for clusters that makes deployment of servers or applications in the cloud easier. It uses cookbooks, which contain recipes, to specify the configuration for each node. A cookbook consists of attributes, definitions, files, libraries, metadata, recipes, resources, templates and tests. On each node, a Chef Client is installed which manages the installation of that node. [19]

2.4.2 Karamel

Karamel is an orchestration tool for big data clusters which offers reproducible deployment of distributed applications on bare-metal, cloud or multi-cloud environments. It aims at being an easily-accessible UI-driven experience for deployment of distributed applications. In Karamel, there is a cluster definition file describing the distributed system as the application stack in the system, the provider(s) for each application stack in the cluster, the number of nodes for each application stack and configuration parameters for each application stack. The application stack is given as Chef cookbooks located at a public GitHub-address containing recipes to be executed. Cookbooks contain a certain Karamel-file specifying its orchestration rules. [20]

The orchestration in Karamel is based around Directed Acyclic Graphs. The installation DAG is created by combining all of the Karamel-files, specifying each recipes dependencies, and out of that build the DAG accordingly. Karamel uses the open-source Chef Solo client for installing the cookbooks specified for each of the groups in the cluster definition file.

Karamel offers reproducibility of clusters which allows for reproducible deployment of experiments and benchmarks in the cloud. The results of experiments can be stored persistently before shutting down clusters.

Karamel uses a VMM as described in Section 2.2.3 on page 11, more specifically JClouds. Since Karamel supports multiple cloud providers a standardised API was required.

2.5 Event Stream Processing

The vast increase of data that is being collected calls for new ways of processing and filtering the data. Systems like MapReduce are store-and-process schema which requires storage capacity sufficient for a large-scale system and in addition might be too slow for critical situations. This has motivated the development of Event Stream Processing (ESP) systems that, in comparison with MapReduce
systems have low latencies and can handle a large amount of data in real-time [21]. Event Stream Processing is generally used to find the most interesting information or patterns in real-time in large sets of data. One example of this is Apache Storm [22], which is used by Twitter among others. Another example is Riemann which is part of the stack that Cloudify uses [23].

2.5.1 Riemann

Riemann is a single-node transient event stream processor. The events essentially consist of keys and values and in Riemann there are some predefined attributes: host, service, state, time, description, tags, metric and ttl. In addition to these, custom fields can also be created, however, they perform slightly worse compared to the standard fields due to the hashmap lookup.

Events are stored in the index and last for the duration of ttl or until they are overwritten. They become overwritten when another event with the same unique key enters the index. The unique key is comprised of the host and service attributes. As events are continuously overwritten, the data is meant to be forwarded or read immediately. Data is often forwarded to a time series database like InfluxDB, sent to other event stream processors or other components. Events can also be sent via email to alert system administrator’s when for example certain thresholds have been exceeded. Another way to acquire data is by querying the index.

As Riemann uses Clojure, a functional programming language, it naturally utilises functions a lot. Events flow into streams, which are functions, and may be filtered to child streams, which are also functions. The events that flow into the same stream share the same scope. Functions like coalesce can aggregate the events that are indexed to a child stream, which allows for computations on data sets that are closely related.

In Riemann events are sent over Protocol Buffers to compress the data and thereby consuming less network bandwidth. It works by a push model, which means that it receives events instead of the alternative polling model where it would request events. Events sent over TCP must first pass through all the layers of Riemann before the sender is notified with an acknowledgement. If the sender is synchronously waiting for an acknowledgement, overwhelming the Riemann server with data is avoided. However, if the sender needs to send events faster than the time it takes to wait for the acknowledgement there are some different options. These options do all, however, have the risk of causing memory overload in the Riemann server. [24]

2.6 Elastic Scaling

Elasticity in distributed systems is defined as ‘the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible’ [25, pp. 3].
Elastic Scaling has over the years been extensively studied and applied and can in principle be derived into three categories: reactive, proactive and mixed approach. Reactive approach scales according to reported changes in workload. The proactive approach, on the other hand, predicts future workload changes and scale the application accordingly. A mixed approach can scale both reactively and proactively. [14, pp. 21-22]

Because of the changes in pricing in cloud computing described in Section 2.2 on page 10 the more unreliable but cheaper spot instances, are now being used reliably via an Elastic Scaling system, primarily for cost-efficiency reasons. There are however a lot of problems related to this, for example the prediction of future changes in spot instances availability. [14]

2.7 Concurrency

Subsystems may share resources and as those resources are accessed concurrently, there is a possibility for race conditions to occur. It happens when a critical section is accessed without mutual exclusion by more than one thread. Mutual exclusion can be attained by using locks, semaphores or busy waiting. Whenever threads are dependent there is a risk for deadlocks, livelocks and starvation. [26, pp.47-56]

Real-time systems that interact with users and other systems requires a non-deterministic program flow, often with many moving parts. Users and other systems are preferably served immediately in a non-blocking manner. Asynchronous thread-safe interfaces are therefore favourable. An implementation of such an interface would be a producer-consumer type scenario. Requests, if of equal importance, are put in a thread safe FIFO queue. Another thread then dequeues the contents and serves those requests accordingly.

2.7.1 Concurrency in Java

Typical race conditions in Java are thread interference and memory consistency errors. The former means that some operation, consisting of several steps, is interleaved by another thread. The latter occurs when threads have an inconsistent view of the memory, a problem related to the consistency model used by the processor. These problems can be avoided by either using synchronized blocks or concurrent data structures. Java has reentrant synchronization on their synchronized blocks. That means that the same lock can be taken numerous times, by the same thread, without risking deadlocks. [27]

Concurrent data structures have a built-in mutual exclusion for critical sections. One example is weakly consistent iterators. They guarantee that the elements of the data structure, upon creation of the iterator, will be traversed through exactly once. Elements that are added subsequently by another thread may also be handled, but it is not guaranteed. [28]

In Java coordination between threads is most often attained with a guarded block. The guarded block calls wait and is later invoked by notify. In the
In a producer-consumer scenario, this approach can be used. Another way is to use a `LinkedBlockingQueue` which blocks until values are present. A `ConcurrentLinkedQueue` on the other hand is non-blocking but still thread-safe. [27]

Some data structures like the aforementioned FIFO queue in the producer-consumer example must be continuously altered. However, when this need is not apparent, an easy way of simplifying parallel programs is through the use of immutable objects. Some guidelines for creating immutable objects in Java are `final` and `private` fields, as well as declaring the class as `final` to prevent inheritance and not using setters [29].

For managing and creating threads the `Runnable`-interface or `Thread`-object can be utilised. However, for large scale applications, the overhead from thread management might become overwhelming. With a `ThreadPoolExecutor` the number of threads can easily be bounded, which helps preventing too many or too few threads to be operating. The `ThreadPoolExecutor` also separates the resource-consuming thread management from the application, thus minimising the overhead from thread creation. [30]

### 2.8 Related Work

Monitoring systems for clouds have been around for a long time and there are a lot of different monitoring systems out on the market. In the year 2014 a survey discussed many of the, at that time, available monitoring systems. The monitoring systems were categorised into three categories: Commercial platforms, open source platforms and services. None of the systems listed as Commercial Platforms nor open source platforms support either resilience, reliability, availability and accuracy. Some of the systems also do not support cross-cluster monitoring since they are tied to a specific provider, such as Amazon CloudWatch and AzureWatch. Only one of all the monitoring systems examined in the survey were considered to be reliable as it is defined in Table 1 on page 15. [1]

#### 2.8.1 Collectl monitoring

Collectl is a system performance monitoring tool with many usages. It can format the data in different ways and has the possibility to collect a lot of different parameters. The data can either be saved to a local file, printed to the standard output or sent over the network via TCP. It is specialised for running on all Linux distributions and does therefore not remarkably impact performance. For collection in one-second interval, it has an average CPU utilisation of 0.1%. [31]

In addition, there are also some different plugins developed for Collectl, most notably Colplot and Colmux. Colplot is essentially used to plot the data coming from Collectl. Colmux is a tool developed for running Collectl on a distributed system where it receives all data from each node and outputs it as a single stream of data. It also has the ability to control all of the Collectl processes running on the cluster machines. The developers, however, emphasise on that
running it on a large cluster will result in a heavy load on the node running Colmux. [31]

In our work, this monitoring system stack was at first considered a possible solution. But as it receives all Collectl metrics and filtering only occurs in the central node it was concluded that it would not be suitable for this work.

2.8.2 Cloudify stack

As mentioned in Section 2.2.5 on page 12 Cloudify has monitoring, amongst many other functionalities, built-in. Cloudify utilises three main components for monitoring: RabbitMQ, Riemann and InfluxDB [32, 24, 33]. RabbitMQ, a message broker, receives all metrics from the distributed agents running each of the cloud machines. From RabbitMQ there is a Riemann server reading metric continuously, performing calculations and then putting it into InfluxDB, a time-series database. [23]

The Cloudify stack is in this work seen as a proof that our architectural design decision are, at least partly, suitable by commercial standards. In Tablespoon there is no message broker and it instead uses Riemann for receiving events from the distributed agents. But the essential idea is similar.

2.8.3 Riemann-InfluxDB-Grafana stack

This stack is capable of both storing the monitoring data within a database, as well as providing real-time graphing. For this project it was, however, not considered a suitable solution, mainly due to Grafana not being easily integrable with Karamel GUI. In addition, having a database; thus storing monitoring data for a longer time period, was not part of the scope of the project.
3 Methods and Methodologies

This section will present and briefly discuss the different methods and methodologies related to the thesis project. In this thesis different evaluation methods have been employed in order to evaluate all of the systems functionalities.

3.1 Research Methodology

Design science is a research methodology within the field of Information Technology. Design includes both the process of creating a product, as well as the created thing, the final product. The purpose of this methodology is essentially to create an innovative product that solves a real problem. Two fundamental questions are therefore stated as: ‘What utility does the new artefact provide?’ and ‘What demonstrates that utility?’.

To answer these questions Hevner et al. describe seven guidelines [7]. The first regard defining the artefact and its validity. To be a valid artefact it has to include a construct; which is the language to express the technology, a model; which uses the construct to present a real world situation, methods; which describe the processes around and within the artefact through mathematical models or best practices, instantiations; which demonstrates the validity of constructs, models and methods. This guideline is fulfilled by describing the created artefact and the practices used in Section 4 on page 23.

The second guideline addresses the problem relevance. What is the underlying phenomena that motivate the research? In what way does the creation of the artefact affect businesses and the constituent community (practitioners who plan, manage, design, implement, operate, and evaluate information systems)? This is discussed in Section 1.1 on page 7 and in Section 6.3 on page 42.

The third guideline represents the evaluation of the design. The evaluation methods are first presented in Section 3.3 on the following page and the artefact is then evaluated extensively in Section 5 on page 32. The features of the created artefact are compared to other, closely related, inventions. The evaluation takes into account in what way the designed artefact integrates with the current technical infrastructure.

The fourth guideline discusses how the artefact may contribute to the research community. Can the created artefact serve as a basis for hypotheses to be tested by future empirical work? What can the new design principles be used for? Apart from being mentioned in Section 7.1 on page 44, the research contributions of the work is not addressed in this thesis.

The fifth guideline involves the research rigour of the created artefact. Experiments that examine the functionality of the artefact are presented in Section 5 on page 32.

The sixth guideline concerns the design process. It answers the question: What was the motivation behind the design choices made? What ends; the goals and constraints on the solution, means; the set of available actions and resources,
laws; the uncontrollable forces in the environment, were affecting the design process? The design process is discussed in Section 6.1 on page 38.

The seventh guideline brings up communication of research. How accessible is the work and who is the target audience? How can it be conveyed to both technology-oriented as well as management-oriented audiences? The communication of research is briefly discussed in Section 6.3 on page 42.

3.2 Code Testing

Tablespoon was developed using test-driven development. This meant continuously writing tests that, in different ways, would test the functionality of different components.

3.2.1 Unit tests

During the development of Tablespoon, unit tests using JUnit4 [34] were often written for the more complex classes. JUnit is a software built for writing unit tests in Java. These tests are meant to test isolated parts of a system to validate that it functions as expected. Since components in many cases are dependent on each other, the tests may have to involve mocking other components. Mocking essentially means that you create a fake object from the actual class and redefine the methods. For instance, if you have a class responsible for communication with a remote server, you could mock that class and instead of sending actual data to the server you just count the number of messages received in the class.

3.2.2 Integration tests

To test the behaviour of several components together, some integration tests were developed. The integration tests main responsibility is to test communication functionality between components. In order to differentiate these tests from unit tests the Maven Failsafe plugin [35] was used. Maven Failsafe plugin uses a special naming convention where tests with names ending with IT are identified as integration test and will therefore not run when a user performs a normal build of the project.

3.3 System Evaluation Method

The system is evaluated based on the 12 characteristics mentioned in Table 1 on page 15. In this thesis, the validation of each of the 12 characteristics is accomplished by qualitative reasoning with the basis in the research method introduced in Section 3.1 on the previous page. In addition to this, some experiments have been conducted in order to give insight on the performance of the monitoring system.
4 Tablespoon in Karamel

Tablespoon has been developed for the specific purpose of providing real-time measurement data from cluster machines to an Elastic Scaler in Karamel. This chapter will present and describe each of the modules developed separately, as well as how it was integrated into Karamel.

4.1 Tablespoon

Tablespoon is a monitoring system for distributed environments which offers easily reconfigurable performance measurements. As seen in Figure 1 it involves three main components: An agent installed on every machine, a server that consists of an event stream processing server and a client that is built-in into Karamel and provides an interface for others to communicate with. In addition to these three main components, there is also a module within Karamel that integrates Tablespoon into the Karamel system.

One of the major design philosophies behind Tablespoon is reconfigurability - meaning that agents can be adjusted to fit the need of the user. If the system is under stress it is possible to only request a small amount of metrics. If this is not a concern, one can subscribe to up to 68 different parameters that are offered by Collectl. The agents also, through the use of promises, helps prevent the server from being overloaded. This aspect along with reconfigurability makes the system more scalable and less prone to failure.

![Figure 1: Overview of Tablespoon architecture.](image)
4.2 Agent

The Tablespoon agent is written in Java and its main purpose is to continuously receive data from a metric gatherer such as Collectl, filter and aggregate information based on the interest of the user, and then sending compressed events to the server-side. The agent blocks while waiting for a promise to be fulfilled. This limits the send rate to the capacity of the Riemann server, preventing it from overloading.

![Flow chart of agent](image)

Figure 2: Flow chart of agent

The agent has two main threads as can be seen in Figure 2, one responsible for receiving the input from Collectl and one for parsing that data into events by Riemann definitions. The thread receiving data from Collectl matches it into Java objects by regular expressions and comparing to predefined expected data. The objects are then put into a queue from which the other thread reads.

As it waits for the queue to fill it perpetually scans the directory intended for topics. A sleep gap was introduced to limit CPU and I/O usage. When a new topic file arrives, it is read, parsed and subsequently removed from the hard drive. Topics are expressed with the JSON format and the jackson-jr library is used for parsing the file. It is designed specifically for data-binding in lightweight applications [36].

The agent uses the Netty-based Riemann Java client for transport [37]. It can either send single messages with RiemannClient, or specialise at sending batches with the RiemannBatchClient. Tablespoon utilises the latter as can be seen in Figure 3 on the following page. If the agent does not get an acknowledgement in riemannDereferenceTime, the agent will attempt to resend the batch provided that the connection is still established.
private void sendBatch(List<Event> batch) throws IOException {
    try {
        IPromise<Msg> promise = rbc.client.sendEvents(batch);
        Msg msg = promise.deref(config.getRiemannDereferenceTime(),
                                  java.util.concurrent.TimeUnit.MILLISECONDS);
        if (msg.hasOk() == false) throw new IOException("Timed out.");
    } catch (IOException ex) {
        if (rbc.isConnected()) sendBatch(batch);
        else {
            throw new IOException();
        }
    }
}

Figure 3: Agent sends batch to Riemann server.

The Riemann Java client also ensures compression of events. It converts a predefined key-value structure to Protobuf objects, a language- and platform-neutral way of serialising structured data [38].

Ensuring thread-safety in the agent was done mainly by choosing a simple sequential program flow and using few shared resources. The only resource that need to be shared is the metric queue. New metrics need to be added, dequeued by another thread and finally packaged into events. Old metrics need to have a retention mechanic since the agent is a daemon that may run for an arbitrary time. If the connection to the server is lost the agent should not keep filling the queue indefinitely. The thread, therefore, dequeues elements when their ttl duration is overdue. The synchronized block encapsulates any code manipulating the metric queue. The only code that is encapsulated are simple add and remove operations, while code that communicates with other systems like the file system, collectl-process or Riemann server runs without such restrictions.

4.3 Server

The server-side in Tablespoon consists of the event stream processing component Riemann introduced in Section 2.5.1 on page 17. Events are if correctly tagged, temporarily stored in Riemann’s index until their time to live parameter is expired. The event types include GROUP_AVERAGE, the mean over the machines belonging to a topic; GROUP_MEDIAN, the median over the machines belonging to a topic; and REGULAR, which represents unaltered events. Before calculating the mean the 10th and 90th percentile is eliminated for a better average value.

The state field is marked with "tablespoon" for every event belonging to tablespoon. This is to separate them from the many other events that Riemann can listen to. As mentioned in the background, each event in Riemann is uniquely identified by the combination of host and service. In Tablespoon, the service field contains the unique identifier of a topic. This implicates that a host may only have one event, from the same topic, active at one time in the index.
In Figure 4 it is shown how the ambition for every event is to reach the index. The state field is the first barrier; filtering out any event without the correct value "tablespoon". The events then pass through the by-function, which creates child streams depending on a field. In this case, the field is service; effectively creating a child stream for every separate topic.

**REGULAR** events reach the index without any modification. The other two event types require new events to be created. The *coalesce*-function gathers all of the events present in a child stream into a vector. This happens periodically every 5 seconds while events still exist in the stream. The *smap*-function is the higher-order-function map, specifically for the metric field in vectors of events. To calculate the median the *median*-function is provided as an argument to the map function. The average is determined by first sorting the list, then removing percentiles from head and tail and then, from the events remaining in the list, calculate the mean.

Lastly, the *with*-function is used to create the new events. It is a useful function for many situations in Riemann since events are immutable. The new event’s host field gets marked with *nil*. It is reasonable since the new event cannot be said to belong to one host. The new event still has the same topic identifier which makes it easy to retrieve from the client.

```plaintext
where state "tablespoon"
    by service
    where tagged "GROUP_MEDIAN"
      coalesce 5
      smap median
      with host nil
      index
    where tagged "GROUP_AVERAGE"
      coalesce 5
      smap custom
      sort
      remove head
      remove tail
      mean
      with host nil
      index
    where tagged "REGULAR"
      index
```

Figure 4: Pseudocode of Tablespoon’s Riemann configuration.

### 4.4 Client

The client module contains the code that runs locally on the user’s machine. It communicates with several other components as can be seen in Figure 1 on page 23.
4.4.1 API

The API is used for making requests about monitoring information. Anyone making a call must supply a `Subscriber`-interface. Through the `onEventArrival`-method `TablespoonEvent` linked to their request is then received. Upon creation of a topic a unique identifier, consisting of a 128-bit value generated by Java's `UUID`-component, is handed to the caller. The identifier can then be used to remove, replicate or replace the topic. All the features of Tablespoon API can be seen in Table 2 on the following page.

A topic is created with the builder pattern. When all necessary parameters have been added a call to the `submit`-method will submit the topic in its current state. Some parameters are mandatory while others are interpreted to a default value if not specified.

Topics are immutable. In order to make changes, topics can be removed and then new ones can be added. This, however, does not guarantee a smooth transition within the agents. An alternative way is through the `replace`-method as can be seen in Figure 5. When a topic is replaced, the agents make an immediate swap, preventing information from being sent more than once.

The second parameter of the `replace`-method is set to `true`. That indicates that the settings of the previous topic should be applied onto the new topic. The name we have chosen for this feature is replicate. On the following line, the user sets a new threshold, raising the value from 70% to 75% before the newly created topic is submitted.

```java
/* preparing resources */
Threshold t1 = new Threshold(70.0, LESS_THAN);
Resource r = new Resource(CPU);

/* first call to the API */
String uniqueId1 = api.submitter().
    subscriber(subscriber).
    groupId("group 1").
    eventType(GROUP_AVERAGE).
    resource(r).
    duration(60).
    sendRate(5).
    high(t1).
    submit();

/* second call to the API */
Threshold t2 = new Threshold(75.0, LESS_THAN);
String uniqueId2 = api.submitter().
    replace(uniqueId1, true).
    high(t2).
    submit();
```

Figure 5: Example showing the construction of API calls.
Table 2: Table explaining features of the API.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscriber</td>
<td>An interface that receives TablespoonEvent upon arrival.</td>
</tr>
<tr>
<td>groupId</td>
<td>Identifier for a group of machines.</td>
</tr>
<tr>
<td>machines</td>
<td>A set of specific machines, can not be used in conjunction with groupId.</td>
</tr>
<tr>
<td>eventType</td>
<td>Can be either GROUP_AVERAGE, GROUP_MEDIAN or REGULAR.</td>
</tr>
<tr>
<td>resource</td>
<td>Can be specified as one of 69 parameters, or more broadly as CPU, MEM, NET or DSK.</td>
</tr>
<tr>
<td>duration</td>
<td>Specifies for how long the topic will be active on agents.</td>
</tr>
<tr>
<td>sendRate</td>
<td>The rate at which agents send events.</td>
</tr>
<tr>
<td>retrievalDelay</td>
<td>An artificial delay added to prevent excessive querying.</td>
</tr>
<tr>
<td>high</td>
<td>A Threshold which filters events based on a percentile.</td>
</tr>
<tr>
<td>low</td>
<td>A secondary Threshold to create two boundaries.</td>
</tr>
<tr>
<td>replace</td>
<td>Is used to replace or replicate a topic.</td>
</tr>
<tr>
<td>submit</td>
<td>Submits the API call asynchronously.</td>
</tr>
</tbody>
</table>

4.4.2 Broadcasting

Topics are stored in a ConcurrentHashMap. It is, therefore, possible for calls to be submitted simultaneously from several different threads. When a new topic is created a JSON message is immediately constructed. The JSON message is then sent to all of the agents that are registered to that topic. As the new topic enters the TopicStorage a guarded block within AgentBroadcasterAssistant is invoked. That makes it leave the waiting state as can be seen in Figure 6.

The AgentBroadcasterAssistant then enters the cleaning phase. It essentially means removing associations with no longer active machines or expiring topics that are overdue.

![Figure 6: A state diagram depicting the states of AgentBroadcasterAssistant.](image)
The following phase regards broadcasting topics that have been explicitly re-
moved. Since topics are immutable, removing topics is naturally another kind of
operation than creating topics, demanding an auxiliary data structure. Topics
scheduled for removal are placed in a ConcurrentLinkedQueue and dequeue in
this phase. As mentioned in Section 2.7.1 on page 18 this queue is thread-safe
and non-blocking.

In the broadcasting topics phase, a weakly consistent iterator of the topics
is created. On each iteration, the JSON message along with the correspond-
ing machines is submitted to the sendToMachines-method. This method is
part of the AgentBroadcaster-interface. An interface implemented by the
ClusterManager-class which resides in Karamel. The following steps are dis-
cussed in depth in Section 4.5 on the following page.

4.4.3 Retrieving Events

Events are brought from the Riemann index to the client by continuously polling
throughout the duration of the topic. Riemann java client, the same component
that is used in the agent, is responsible for querying. The author of the client
explains: ‘Each client allows thousands of outstanding concurrent requests at
any time, so a small number of threads can efficiently pipeline many operations
over the same client.’ [37]. For this reason, one RiemannClient together with a
pool of worker threads is used. They are needed because a query blocks until it
receives an answer. Without a pool, latency over the server link would severely
impact subscribers of other topics. The ThreadPoolExecutor is set to a static
capacity of 16 threads, which was based on the latency experiment shown in
Section 5.2 on page 33.

Since the hosts are distributed, the Riemann index will not be entirely syn-
chronised in most cases. This leads to a few problems the client has to deal
with.

One problem is that events will be retrieved more than once, which happens
whenever events are not updated between queries. It is preferable to never
forward the same event more than once to a subscriber. It is dealt with by
having a LinkedHashSet in every separate fetcher. LinkedHashSet works as a
FIFO queue and retains a memory of the last 2000 events that passed through
there. To be able to ensure the set property of the data structure, the equals-
method of the TablespoonEvent-object is extended.

Another problem is when an overwrite happens too suddenly before the event
has been fetched. If an event is never fetched it will be lost in the current
implementation. The probability of that happening changes with the rate of
querying along with some other factors. This model represents this occurrence:

\[
P(\text{loss}) = 1 - \frac{t_{e_2} - t_{e_1}}{t_{q_2} - t_{q_1}}
0 \leq t_{e_2} - t_{e_1} < t_{q_2} - t_{q_1}
\]
\[
t_{q_1} < t_{e_1} < t_{q_2}
\]

29
\( t_e \) represents a point in time when an event arrives at the Riemann index. \( t_q \) represents a point in time when a query from the client reaches the Riemann index. These values will vary as they are dependent on network stability, latency and their corresponding rates; send rate and querying rate. When the send rate or querying rate decreases, it is more likely for the events or queries to arrive close to each other. In the model it is certain that \( t_{e1} \) occurred between \( t_{q1} \) and \( t_{q2} \). If \( t_{e2} \) also occurred within that time frame, then it is certain \( t_{e1} \) has been overwritten before it was read. Whenever \( t_{e2} - t_{e1} < t_{q2} - t_{q1} \), there are risks for data loss. If data loss is a concern, a low enough querying rate should be chosen.

In order to prevent excessive querying the `retrievalDelay`-option is included in the API, as seen in Table 2 on page 28. If not specified, the default querying delay is 50% of the send rate. This means that timeliness of events will decrease when the send rate is lowered. We assume that in most cases, events with a lower send rate also have less urgency. In this way we limit the amount of events being retrieved more than once, while still satisfying \( t_{e2} - t_{e1} > t_{q2} - t_{q1} \) when no erratic delay between the client and server, or host and server, exists.

### 4.5 Integration with Karamel

As described in Section 2.4.2 on page 16 Karamel uses directed acyclic graphs (DAG) for scheduling tasks on the cluster machines. The Tablespoon agents are controlled in a similar fashion. During the cluster installation phase, if Tablespoon was enabled in the cluster definition file, Tablespoon will be installed. In the cluster definition file the user also specifies an instance on which the Riemann server is to be installed. The installation is managed by a Chef cookbook developed for this project. The agent installation is separated into two dependent tasks; firstly Collectl is installed, thereafter, if Collectl installation was successful, the Java agent is installed.

Some of the architecture of Karamel was altered in order to enable Tablespoon to function as planned. In Karamel, cluster machines belong to groups and as the Elastic Scaler scales a cluster, the group composition may change. Since Tablespoon supports filtering and aggregating events based on groups it has to be continuously updated with the current state of the cluster groups. Inside Tablespoon, the `Groups-object`, therefore, has a `addMachine`-method which is used by the `ClusterManager` in Karamel.

The Riemann node is located in the cloud. Running Riemann on the Karamel machine could interfere with the user’s firewall settings. It would require opening up a server socket to which all the clients connect. To avoid this security compromise, the Riemann server is deployed on one of the virtual cluster machines via a karamelised Chef cookbook. The users can, in the cluster definition file, specify a virtual machine for the Riemann server to run on.

A feature that is essential for the integration of Tablespoon into Karamel, is prioritised queues into the cluster machines. As of yet, each cluster machine has one queue into which Karamel puts tasks and the cluster machine then executes them in the incoming order. However, since the tasks controlling Ta-
blespoon agents are of much higher priority, a separate queue needs to be created into which all Tablespoon-related tasks should go. This queue should always be firstly read in the cluster machine to ensure no unnecessary delay for the monitoring system.
5 Evaluation

In this section a set of experiments are presented. They illustrate the performance of Tablespoon under various circumstances. It is followed by an qualitative evaluation using the twelve cloud monitoring characteristics introduced in Table 1 on page 15. The evaluation is supported by the results of the experiments.

For the evaluation a set of parameters have been identified. These parameters are categorised into either environment parameters or system parameters. The system parameters are controlled directly by the system, as opposed to the environment parameters over which the system has no direct control. The environment parameters consist of the number of topics and their configuration, the frequency of events that exceed a threshold, the number of agents, send rate and I/O latency. A system parameter is for example the size of the thread pool used for querying the Riemann index. The aim of this section is to, at least partly, prove that the goals stated in Section 1.3 on page 8 are met. The goals are primarily to minimise latency and resource usage.

5.1 Use Cases

In order to make realistic assumptions, typical ways of using the system needs to be extrapolated. Two Use Cases, that are meant to come bundled with Karamel, are the Elastic Scaler Use Case and GUI Use Case described in the bullet list. The complete Use Case is a likely configuration if system administration is a high priority or if the data is saved to a time series database, in order to analyse and make critical decisions over large data sets. The System Administrator Use Case may be used by someone who wants to keep track of the long term performance of their instances, but has no urgency in making decisions about that data.

- **Elastic Scaler Use Case** - Subscribing to the 4 parameters CPU, MEM, DSK and NET at certain thresholds with the event type REGULAR and send rate 1. If all thresholds are exceeded at the same time the agent will send 4 events every second. Additionally, a time bound subscription to the GROUP_AVERAGE is set up sporadically.

- **GUI Use Case** - Subscribing to 15 different parameters that sends every 10 seconds. Messages are always sent, no thresholds are applied.

- **Complete Use Case** - Subscribing to all the 68 parameters that are offered by Collectl with send rate 1.

- **System Administrator Use Case** - Subscribing to 10 metrics. The send rate is set to 3600 which means that the average of 3600 samples are sent every hour.
5.1.1 Demonstration

The different use cases performance is demonstrated by testing the CPU and network usage. The values are gathered via Collectl. Each configuration was run for 10 minutes collecting the metrics concurrently. The demonstration was run on Amazon Web Services t2.micro instance (1 vCPU and 1 GiB memory). The result are shown in Figure 5.1.1.

![Comparison between different user scenarios](image)

Figure 7: Comparison between different user scenarios

5.2 Latency Experiment

The experiment concerns the timeliness of Tablespoon. It measures the time from when a metric is produced by Collectl to when it is read by the subscriber. This latency depends on a number of factors: The time it takes for the agent to parse the metric into protobuf, the latency between the agent and the server, the time it takes for Riemann to interpret the message and place it in the index, the latency between the server and the client, the point in time in which the client decides to query the server and lastly how long it takes for the subscriber to receive the message within the client.

In this experiment the total latency of all the above steps are measured. However, since the experiment runs locally there is no network latency between any components. The varying factor in the experiment is the number of threads that are querying Riemann server. A thread will block until it receives an answer, during that time other threads can make queries simultaneously. That is if there are threads available in the thread pool. If there are no threads available the client will have to wait. To illustrate this waiting time, an artificial delay is added. This makes the difference between having single or multi-threaded querying more apparent.

Three agents are started on 3 separate Vagrant virtual machines. The agents are configured to run the Elastic Scaler Use Case. No thresholds are applied on the events, meaning that each agent sends 4 events every second. The GROUP.-AVERAGE event type is not part of the experiment as it received infrequently.
In this experiment data was overwritten several times as can be seen in Figure 8.

![Graph showing latency results](image)

Figure 8: Graphs showing the results of the latency experiment

### 5.3 Evaluation of Cloud Monitoring Characteristics

In this section a qualitative evaluation of the twelve monitoring characteristics and how well Tablespoon is fulfilling them is presented. The degree to which each of them is fulfilled is summarised in Table 3.

<table>
<thead>
<tr>
<th>Monitoring Characteristic</th>
<th>Fulfilled</th>
<th>Partly Fulfilled</th>
<th>Not Fulfilled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elasticity</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptability</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Timeliness</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Autonomicity</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehensiveness</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensibility</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Low intrusiveness</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Availability</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accuracy</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Scalability

One argument for scalability is that filtering is being done on the agent-side. This limits unnecessary events from being sent over the network. The other argument for scaling is that the agents use promises and will, if the promise is rejected, notice if an event is not processed by Riemann. This interleaves the frequency of events which helps prevent the server from being overloaded. The Riemann server is a single point of failure, meaning it will become a bottleneck under extreme circumstances. Therefore, Tablespoon is only partly scalable.

Elasticity

Karamel maintains a SSH-connection with all the instances on which the Tablespoon agent runs. When the connection is lost Karamel will notify Tablespoon. Tablespoon reacts to this occurrence by no longer forwarding information to that machine. As a new instance is being created, Karamel deploys Tablespoon agent and starts monitoring. When Tablespoon is implemented together with Karamel elasticity is thus achieved.

Adaptability

Tablespoon is not adaptable, as it does not adjust to the workload of the virtual instances. Instead, this task is assigned to the subscribers using the API.

Timeliness

The timeliness in Tablespoon can be described as the latency between when the metric is read in Collectl and until it arrives at the subscriber. In Figure 8 on the previous page the average latency for an event is shown, dependent on the thread pool size. In the Elastic Scaler Use Case the expected latency is around 500-700 milliseconds. As the timeliness of a monitoring system is dependant on the requirements of the users, Tablespoon is timely in some of the Use Cases.

Autonomicity

Because Karamel can handle machine instances failing, Tablespoon becomes partly autonomous. But in every other regard, such as adjusting to the workload of a machine instance, Tablespoon is not autonomous.

Comprehensiveness

Tablespoon is focused on bringing system level metrics. In this particular area it provides 68 parameters offered by Collectl, along with one custom parameter that was developed specifically for Tablespoon. Application level metrics are not supported, nor is metrics related to network latency between hosts. The conclusion is that Tablespoon is comprehensive for its intended use, as Karamel has no knowledge of the application layer. Although as a monitoring system as a whole it is only partly comprehensive.
Extensibility

Tablespoon does not support plug-ins. It is not tightly coupled with Collectl and could therefore be extended in a future release. In this state however, the monitoring system is not extensible.

Low intrusiveness

Agents only send events that have been explicitly requested, thus limiting an extensive amount of unimportant events from being sent. This restricts the intrusiveness on the instances I/O and the cloud network. Moreover, Protobuf compression and the fact that events are sent as a batch rather than individually, result in less strain on the network.

When running the 4 different Use Cases presented in Figure 5.1.1 on page 33, the Tablespoon agent’s average CPU utilisation was always below 0.5 %. In the Elastic Scaler Use Case, the I/O on the network was an average 1.0 Kb/s.

Riemann uses in-memory architecture and does not save events. Therefore, the server does not take up extensive disk space. Tablespoon exhibit low intrusiveness.

Resilience

When evaluating the resilience of Tablespoon, each component is reflected upon individually. For example, if Riemann fails the user will not receive any monitoring information during that time. However, if Riemann is restarted, the agents will reconnect automatically. While the agents are disconnected they still gather data. They also remove expired data in order to not overburden the memory.

Tablespoon, however, makes no estimations about whether or not an agent has failed. If an agent fails, data from that particular node will be lost and the user of the API is not notified. Currently it is up to the user to notice if a particular node stops responding. One reason for the agent to fail is if Collectl fails, as it is dependent on Collectl for collecting monitoring information. Within the agent there is no mechanism for handling a failure in the Collectl process. However, if an instance fails it will be handled by Karamel, as was mentioned earlier.

Regarding Riemann, it is a single point of failure component and there is no feature for bringing the server up again. Besides, the client has no knowledge of whether or not a query was unsuccessful and does not adjust to such situations. In conclusion, Tablespoon can not be stated as a resilient monitoring system.

Reliability

The agent prevents memory leakage and their performance, in terms of CPU usage, was below 0.5% during the experiments presented in Figure 5.1.1 on page 33. Since data is only kept in a hash map in Riemann, sometimes data is
lost before it is retrieved. If these events happened to be highly important to the subscriber, the system can be seen as less reliable. For this reason, Tablespoon is only partly reliable.

Availability

Agents adhere to their topic configuration, changing immediately, providing high availability of data. The API is thread-safe and non-blocking, therefore topics can be created at any time.

Accuracy

Tablespoon accuracy is to some extent dependant on the accuracy of Collectl. Apart from that, there are also other factors that affect accuracy. For instance, if the agents are heavily stressed, it can not be expected to provide accurate data.
6 Discussion

This section discusses Tablespoon with an emphasis on the evaluations of it. The discussion begins by reasoning about the design process and fit of the organisation in line with what was described in Section 3.1 on page 21. It continues by considering research aspects and concludes with a more general discussion of the system.

6.1 Design Process

Tablespoon was, during its development, continuously refined to meet the needs of Honeytap, an Elastic Scaler. One major benefit was that the development of this component was going on at the same time, allowing for interdependent discussions and design choices. Throughout the process we had close correspondence with the developer of this component, as well as with our supervisor who was, at this time, the project leader of Karamel, ensuring its fit with the ecosystem and design principles of Karamel.

Prior to this work, Karamel used a simplistic batch monitoring system, not able to fulfill the needs of Honeytap. Therefore, Karamel expressed the need for a real time monitoring system. Another incentive was to give users an overview through a graphical user interface. This overview would come bundled with Karamel, therefore, being easy to set up, and help users verify that the cloud followed the SLAs. The implication of this was that Tablespoon needed to serve more than one type of user. In Design Science Research terminology, this was a law that Tablespoon had to accommodate. Something that also can be described as a law, was that Tablespoon had to be deployable through Karamel. This essentially means that Tablespoon should work on all the architectures that Karamel works on.

6.1.1 Architectural choices

Tablespoon relies on a number of components and libraries. The area of monitoring systems was briefly investigated before choosing the general system design with agents, a server and a client. For each of these layers different components and libraries were considered and below follows reasoning behind the most important architectural decisions.

The previous monitoring system in Karamel used Collectl as the tool to provide system performance metrics. During the process several other similar systems were considered, such as Collectd, Munin and Telegraf. In the end, Collectl was chosen for its flexibility and low CPU usage. It can run either as a daemon or as a process. When running it as a daemon it continuously writes the metrics to a file. This could be beneficial in cases where the data is read sparsely. In this project, it was chosen to run Collectl as a process, piping the output into the agent.

As we identified the need for a central processing component the choice fell on Riemann. This was mostly due to noticing that other monitoring systems were
using it, notably Cloudify and Riemann-InfluxDB-Grafana-stack as mentioned
in Section 2.8 on page 19. It could perform all the more advanced features that
had been requested by Honeytap, such as calculating the average over a whole
group of machines by firstly removing the outliers. The main issue was that it
was written in a completely new language to us, Clojure.

6.1.2 Push-based agents

By having the agents push metrics to the central server, instead of polling the
metrics from the other way, less workload is, in theory, put on the Riemann
server. This also means that the Riemann server does not need to keep a
registry of all the agents currently running.

The Tablespoon client is, however, polling the Riemann server’s index. While
pushing the data from Riemann to the client has some benefits, in particular
eliminating the timeliness synchronisation problems related to querying presen-
ted in Section 4.4.3 on page 29, a polling mechanism was chosen. The remote
querying built into the Riemann client allowed making queries into the Riemann
index and then retrieving a set of data. If it instead allowed opening up a con-
tinuous stream of data, such an option would be interesting to explore.

In the beginning, it was considered running Riemann on the user’s machine.
However, during the process two problems related to this became apparent.
Firstly, if the user’s machine became overburdened, it would render Tablespoon
and Karamel unusable. Furthermore, the push architecture of the agents is
probable to have firewall issues, as the firewall only allows trusted sources to
connect. To have Tablespoon work properly, the user would have to open up
a server socket which would require extra effort and also present a security
compromise.

6.1.3 Publish-subscribe pattern

As the primary goal of this degree project was to provide system performance
metrics to Honeytap, much of the initial architecture was decided based on its
requirements. One of the requirements that Tablespoon had to conform to was
the ability to seamlessly change the type of metrics it receives, the thresholds
and the send rate. To achieve this, Tablespoon is using a publish-subscribe
pattern.

In this implementation of the publish-subscribe pattern subscribers creates their
own topics. They then have the power to, on their own, change the metrics they
receive by replacing the topics. The motivation behind this design was that
Honeytap will often change the required metrics, up to every minute. However,
Honeytap is expected not to subscribe to many different topics simultaneously,
meaning that most of the metrics collected never will be used. Because of this,
each agent filters the data based on the currently active topics, only sending
events belonging to an active topic. In this way, Tablespoon lessens the overall
stress on the network.

39
6.1.4 Scalability

As seen in Section 5.3 on page 34 Tablespoon can only be said to be partly scalable. The motivation was that the Riemann server is a single node and in a large scale system it is bound to become a problem. During the design process it was discussed to use Riemann in a hierarchical way, meaning that, for instance, each cluster would have one Riemann server and each would be connected to the same central server. If events first could be summarised in the lower tier, the high tier central server would receive less events, as well as having far less incoming connections.

Since we have not been able to run Tablespoon in a large scale system, it is difficult to know at what point the single-server architecture starts to become a problem. But because of the workload distribution in Tablespoon, where the agents practically filter out all unimportant events, scalability should usually not be an issue in many of the cases.

6.1.5 Filtering

One major design aspect was that Tablespoon had to support putting thresholds on events. This to limit the amount of messages being forwarded to subscribers, as well as acting as a notifier, waking up Honeytap when it needed to take action. In many cases, like with Colmux (described in Section 2.8.1 on page 19), agents send a lot of metrics to a central component. The filtering then takes place inside of that component. With Tablespoon it is possible to send a large amount of metrics, as with the Complete Use Case (described in Section 5.1 on page 32). However, in Tablespoon the agents single-handily manage the thresholds.

Initially it was planned to have two filtering-according-to-threshold stages, one in the agent-side and one in the Riemann server. This would make reconfiguring a quicker process when the threshold was raised, as only the Riemann server and none of the agents, would have to be updated. Although when the threshold was lowered, the agents would still have to be reconfigured. In order to prevent this from happening too often, the agents would keep a slightly lower threshold than what was specified in the topic.

Riemann has good capabilities for filtering events according to thresholds. Changing those thresholds dynamically is possible, but Riemann does not present a procedure for doing this in their API. It could be done by updating the contents of the Riemann configuration and then reloading it. It could also be achieved by creating some event type that the Riemann server can recognise, read and from that information, dynamically reconfigure itself.

6.1.6 Immutable topics

In the end it was decided to make topics immutable, rendering them unable to change. The topic exists in several forms on several components: as a builder object in the API, as JSON when being sent over SSH, as a child stream in Riemann and as objects with different functionality in client and agent module.
With immutable topics constructing and managing these objects becomes less complex and the design becomes more coherent. The topic representation in the client module also needs to be accessed concurrently by different API calls. As described in Section 2.7.1 on page 18 making objects immutable makes them less prone to thread conflicts.

Another reason for making topics immutable is how the CAP theorem works. In a distributed system it is not possible to have a consistent and available system at the same time. When agents are reconfigured, should they wait for an acknowledgement that all other agents also have been reconfigured or should they start immediately? It is less costly and more available to start immediately. If topics were mutable they would be presented to the subscriber in an inconsistent state. To still bring events in a consistent manner would require extra checking on behalf of some other module.

6.2 Fulfilment of Goals

In the beginning of the degree project a set of goals described in Section 1.3 on page 8 were formulated. These can, in essence, be described as building a system that fits well into the design principles of Karamel and serves Honeytap with timely and necessary performance metrics. Some of the experiments conducted, as well as the evaluation and code testing, may speak for whether the different goals have been achieved. However, Tablespoon has yet to be tested by real users in a large-scale environment. For this reason it is uncertain whether the same conclusions could be drawn in such environments.

A goal was that the user should be able to configure, and also reconfigure, the system to whichever settings preferred. This is achieved by exposing an API with several options. The rate at which events are sent by agents is one such option, as well as setting specific thresholds. It also lets the user specify exactly which Collectl parameters that should be parcelled into events and sent over the network. This enables the ability to leave a very thin performance footprint if desired. Any of these settings can be changed at any time, making the API flexible from the subscriber perspective.

However, the API lacks flexibility in some regards. For instance, the Riemann server is static from the user’s point of view. The user could update the configuration file of Riemann, that would however require some extra technical expertise. In this version of Tablespoon, the percentiles removed before calculating the mean, are fixed to the 10th and 90th percentile. Furthermore, it is not possible to get a snapshot of the performance at lower rates than one second. For instance, if a user specifies a send rate of 30 seconds, the metrics collected during those 30 seconds will be summarised and sent as an average value.

The goal of ensuring low performance impact, was split into three different performance goals: agents not obstructing other work, sparse network usage and no memory leakage. The first goal was targeted by ensuring thread safety. As for the second goal, there are several factors affecting this. First of all, the choice of handling a lot of the filtering on the agents meant that fewer events would be sent. The same reasoning also goes for the chosen publish-
subscribe pattern which led to events only specifically asked for being sent over
the network. Lastly, as seen in Figure 3 on page 25 the agents send events
in batches, meaning that all events currently in the sending queue are sent
 together. As for avoiding memory leakage the events in the queues are emptied
if their ttl has expired.

Another ambition was to have good code quality, version control, build auto-
mation and unit testing. Throughout the development we attempted to use the
test-driven development (TDD) procedure. Bugs were solved and new features
developed with the help of unit tests. Whenever building the application, these
tests runs automatically, ensuring that those parts are working as expected.
Using TDD also benefits code quality as it is easier to test methods that are
simple in its design [8, pp.110].

When writing the code base of Tablespoon, modularisation was important. As
expressed in [8] separate concerns should be handled in separate modules. For
instance, the broadcasting package contains all classes responsible for sending
topic definitions to agents, as well as those responsible for sending events to
subscribers. In this particular case, it could be argued that this package contains
too much functionality and if the program were to grow this package could easily
be split. In order to make modules less coupled, we introduced interfaces like
SubscriberBroadcaster, which lets modules interact without gaining access
to unnecessary features.

As previously mentioned, it is important that Tablespoon should work on all the
architectures that Karamel works on. At this point, this is partly unknown, as
it has only been tested on Linux and Mac OS X and not for any of the Windows
versions.

6.3 Problem Relevance

Because of the different payment models explained in Section 2.2.2 on page 11,
there can be a significant gain, both in terms of money and performance as was
shown in Section 2.2.5 on page 12, for a customer to have a system capable of
managing provisioning and deprovisioning of instances by itself. The reason for
this is that the system can detect a bottleneck instance in the cluster and find a
new more suitable instance to run the application on. Another possibility is to
use the much cheaper spot instances as was mentioned in Section 2.6 on page 17.
But for a system, in this case Honeytap, to be able to perform these optimisation
it needs to have the proper information available. It, for instance, has to know
the performance of a set of instances as well as the prices for different types of
instances.

For the performance information Honeytap relies on Tablespoon, which lets
Honeytap describe the type of metrics from the instances it needs and also from
which instances they are requested. Tablespoon does however lack the ability
to measure and provide the network latency between different instances, which
is useful for applications with much interaction between instances.
The target audience for this thesis is other students within the field of Distributed Systems. Other stakeholders include but not limited to researchers of Distributed Systems or adjacent disciplines and businesses interested in monitoring systems for distributed system. Because Tablespoon is a open-source project, it is easily accessible to the audience.
7 Conclusions

- A light-weight monitoring system can be built using in-memory architecture and agent-side filtering.

- Requested event loss can be minimised by using several threads for querying.

- Topics can be reconfigured from a user’s point of view and still be consistent throughout the system. This is achieved by keeping topics immutable and making transitions between topics in agents.

- An agent, that collects metrics from collectl, that sends all 68 collectl parameters to a Riemann server once a second, can run on Amazon Web Services t2.micro instance (1 vCPU and 1 GiB memory) utilising an average of 0.5 % CPU.

7.1 Future Work

In a large multi-cloud environment we speculate that the non-scalable Riemann server can become a bottleneck. For future work, we, therefore, imagine implementing, for instance, a hierarchical set of Riemann servers. Moreover, in the current version of the system, almost all of the filtering is accomplished in the agents, this may, however, become a problem if the subscribing components change their topic specifications very often. In these cases, it would be more appropriate to re-balance the filtering responsibility towards the Riemann server. This translates into adaptability as defined in Table 1 on page 15 and is something that should be improved. Another important feature for future work is the ability to measure the connectivity between instances. As an important feature for an Elastic Scaler, this should be supported in future versions of Tablespoon in order to increase its usability.

As mentioned in Section 1.6 on page 9 one important feature for Tablespoon is to have a graphical user interface in which users can visually, in real-time, see the cluster machines system utilisation. This would contribute to giving the user more insight on what is happening in the distributed system. The user interface should be easily understandable and not provide information not relevant to the user.

For making the work more accessible to others, a user manual for Tablespoon explaining how to use it and giving some examples of different use cases, should be created. Regarding the code quality there are also some remaining improvements to be made, such as a complete Java Documentation, more describing logging when debugging and a GitHub Readme to describe Tablespoon to whoever reads the GitHub-page.

Furthermore, in the current version of Tablespoon, a lot of metrics are usually thrown away, mainly on the agents but also inside Riemann. The time to live for metrics is normally no more than 60 seconds. The reasoning behind this is, as mentioned in Section 4 on page 23, to not overextend the network when an agent
reconnects after being disconnected for a while. Additionally, since Riemann only keeps the latest event for each topic-machine pair, the history available at Riemann is non-existent. Meaning that if the user wants a performance history of a cluster, the user would have to store the metrics locally. For future work, we therefore, imagine using InfluxDB next to Riemann. This would give the possibility for longer time storing of metrics. There are some other issues related to this, though, because since the agents toss most of the metrics, those never even reach Riemann, resulting in a very inconclusive database.

To prove Tablespoons validity as a monitoring system it must be tested more extensively. Moreover, in order to prove its characteristics as a monitoring system described in Table 1 on page 15, it should be compared to other existing systems to see if it has some value to other systems than Karamel. In this thesis, the experiments performed do not cover all possible and interesting areas to test. For instance, a similar test to that was shown in Section 5.1 on page 32, where the agents CPU and network performance was tested, should be conducted for the Tablespoon client.
References


