Visualizing Computational Waste in Cloud Computing

Ashton D. Spina (s2906279)

Advisor: Dr. Vasilios Andrikopoulos
and
Dr. Mircea Lungu

Faculty of Science and Engineering
University of Groningen

This dissertation is submitted for the degree of
Bachelor of Science in Computing Science

August 2018

Abstract

This work introduces background knowledge regarding Cloud Computing and presents the concern for the lack of non-proprietary tools to monitor cost and waste in a generalized manner on cloud deployments in the Virtual Machine as a Service (VMaaS) delivery model. It then explores related works on Cloud Monitoring and describes already existing similar tools. Requirements based on the problem at hand as well as the literature are elicited and a service based on probes and an aggregator is designed. Finally, a tool for monitoring cost and waste in the VMaaS service delivery model is presented to the reader and evaluated against the requirements. Lessons on Cloud Monitoring tools and future work are presented to the reader as a result of the evaluation.

Table of contents

List of figures

Chapter 1

Introduction

1.1 Problem Definition

1.1.1 Background Information

Cloud computing is an area of technology that is seeing considerable gains in usage among businesses and individuals seeking to avoid the high startup costs of local computing [4]. The NIST definition of Cloud Computing describes the concept of Cloud Computing best by defining it as, "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." The definition goes on to indicate that this cloud model is composed of five essential characteristics, three service models, and four deployment models. Of particular importance are the three service models known as Software as a Service (SaaS), Platform as a Service (Paas), and Infrastructure as a Service (IaaS). [16] In practice, IaaS is often provided via a subtype of IaaS, Virtual Machine as a Service (VMaaS). This is providing the use of what are known as *Virtual Machines* (VMs), defined as, "a software computer that, like a physical computer, runs an operating system and applications." Virtual Machines have configurations and run on physical machines whose resources are managed by a host [24]. Their usage on public clouds is provided for a fee with a variety of pay structures. Computing Clouds can be private, as in owned and managed by a private company and have resources only available to that company and those they allow to use it, public where a company allows use of the cloud to the public, usually for a fee, or hybrid, being a deployment to a mix of public and private clouds.

1.1.2 Problem

As cloud adoption rises so does the total cost to users. This means optimizations in terms of cost and the waste-reduction that that implies become more significant in their effects. According to RightScale's State of the Cloud Report for 2018, "Optimizing cloud costs is the top initiative again for the second year in a row for all cloud users (58 percent), which is an increase over 2017 (53 percent)"[19]. This establishes a need for a tool that aids in monitoring cloud systems and identifying waste. Waste in the context of this project refers to the *unnecessary provisioning of Virtual Machines*, and can be *identified by below an acceptable threshold utilization of a VM for a sufficiently long time interval*. In addition, in clouds, public or private, monitoring is essential to maintain the health of the deployed application, benefiting both providers and consumers [3] [8] [6] [17] [10]. This further supports the need for a monitoring tool because minimizing waste is beneficial as well.

If cost and thus waste should be minimized then a metric for waste and a value for cost needs to be defined. Once those parameters are defined, the question becomes, "How to instrument the monitoring and visualization of cost and waste of applications deployed in the Virtual Machine as a Service delivery model?" Specifically, the waste and cost of applications deployed on cloud computing services need to be monitored so their waste can be measured and visualized. By these means further action can be taken to lower waste and produce cost saving.

1.2 Scope and High Level Solution

Despite this established need for a monitoring tool that focuses on cost and waste, no known tool addresses this issue. There exists a multitude of papers and literature exploring and surveying cloud monitoring in different aspects [8] [1] [15] [10] [6][17] [13] [27] [2]. As far as the extent of this research can deduce, none of these explore generalized, non-proprietary, waste-focused cloud monitoring. Therefore, a problem can be found in this absence.

In order to solve this problem with the generalized solution that it requires, a definition of minimum functionality is required. A solution to this problem should be designed work on any public cloud (or at a minimum any popular public cloud). This means not only should the solution be deployable on any public cloud, but also that deployment is possible on multiple different public clouds simultaneously. The solution collects data through what will be called a *probe* and this *probe* should both have low overhead and be non-invasive to the the cloud deployment being monitored. This means that resource usage of any probing solution should be low in volume. Any *monitoring solution*, in this context meaning a method for monitoring Virtual Machines, would itself increase utilization of the monitored instance and as such

would affect the measurement of waste as well as potentially affecting the application being monitored. High resource usage and invasiveness to the deployment being monitored would defeat the purpose of the solution, considering the focus of the problem. Any *monitoring solution* in a cloud deployment should be configurable. It should ideally allow the user to configure things like resource-usage, categorization of data, etc. such that a unique solution is possible for any user. The part of the solution called the *aggregator* should process data in a way that all monitored instances should be distinguishable from eachother such that multiple deployments are possible and the *monitoring solution* can distinguish not only that one instance is different from another, but that an instance which is being monitored on a certain cloud deployment should have its data stored separately from any other cloud deployment's monitoring data. This would be essential to properly monitoring and visualizing the cost and waste of deployments.

For the purpose of the project, the *probe* should be compatible with any virtual machine setup with a network connection. Of course, it isn't necessarily possible to test all possibilities so evaluation of *probe* success will focus on the main cloud providers: Amazon Web Services, Google Cloud, and Microsoft Azure. Although, theoretically compatibility should be possible with any cloud provider that hosts Virtual Machines with common Linux Distributions. The reason for a desire for compatibility with many cloud providers is the current diversification of clouds used by most users on the web right now. According to RightScale, "Companies using almost 5 public and private clouds on average"[19]. This means that any tools that's monitoring public cloud hosted instances has to treat the use of a hybrid cloud for deployed applications as a real possibility. As well, the emergence of Cross Platform APIs and the need to use the features of exposed by multiple different cloud providers shines a light on the demand for multi-cloud monitoring [21] [9]

1.3 Contributions

. This project presents a solution to the problem of monitoring Virtual Machine instances in a way that their cost and waste, in addition to standard metrics of monitoring data, can be recorded and visualized for users. The contribution of this project to the body of scientific knowledge stems from the current lack of tools which meet the requirements of being: non-proprietary, monitor Virtual Machines without regard for their host, and provide a focus on cost and waste. In this way, not only will it be explored whether or not this can be achieved in a manner that is in-line with what can be expected of a *monitoring solution*, but also contributes as an example should others wish to create *monitoring solutions* with other foci. To this end, materials related to this project can be found in the form of:

- A code repository with supporting documentation located at https://github.com/a-d-spina-student/waste-cloud-computing. This repository contains all code related to this project.
- An example of the project hosted at https://www.universalcloudmonitoring.com/.
- A thesis paper explaining and evaluating the project created.

1.4 Outline

This paper is final deliverable thesis manuscript consisting of:

- A summary of the state of the art on cloud resource monitoring with a focus on waste and cost, including scientific papers as well as descriptions of current monitoring solutions.
- An elicitation of requirements, system specifications, and evaluation of alternative design options.
- A prototype implementation of the designed system and the accompanying testing that it requires.
- Experimental evaluation and prototype updates and changes.
- A summary of the main findings of this work and identification of open issues as future work.

Chapter 2

Background-Related Work

2.1 Literature

This literature review will cover four papers:

- Cloud Monitoring : A Survey [1]
- Costradamus: A Cost-Tracing System for Cloud-based Software Services [13]
- Modeling and Managing Deployment Costs of Microservice-Based Cloud Applications [14]
- About Monitoring in a Service World [18]

These were chosen as relevant literature for their focus on either monitoring of cloud deployed systems or a focus on the tracking of cost. The intention is to summarize their most relevant points of discussion with reference to cloud monitoring. In this way, a baseline for cloud monitoring and cost-tracing systems can be established.

2.1.1 Cloud Monitoring : A Survey

Cloud computing is used for internet-based services at an increasing rate. This increase is driven by the affordability and ease-of-use of Cloud Computing services. Because of this increase, the complexity of software deployed on the cloud is also increasing to match. Monitoring of these systems is essential to maintain them and therefore a monitoring solution becomes necessary. Monitoring cloud-based systems is a less-developed field in comparison to monitoring traditional systems because of the remote nature of the systems themselves. The article seeks to discuss why Cloud Monitoring might be needed and then also define the metrics which might be monitored. These metrics might include:

- Computation Based metrics such as:
  - Number of requests per second
  - Central Processing Unit (CPU) Speed
  - CPU time per execution
  - CPU occupation of a Virtual Machine (this is useful when monitoring many Virtual Machines)
  - memory page exchanges per execution
  - disk/memory throughput
  - throughput / delay of message passing between processes
  - duration of specific predefined tasks
  - response time, VM startup time or other VM related timings.
- It may also include Network-based metrics such as:
  - round-trip time
  - jitter
  - throughput
  - packet loss
  - available bandwidth
  - capacity
  - traffic volume

The article discusses that Cloud is more complex to monitor than traditional server farms (Grid Computing), mostly because of the level abstraction that is necessary when providing Cloud Computing services to a customer. It eventually goes on to discuss some current (as of early 2013) services that seek to achieve a result similar to what is described in the article. Considering the pace of technology the 5 year difference (2013-2018) makes this comparison obsolete. The article elicits a few important requirements for a cloud-based monitoring system. It must be: scalable, elastic, adaptable, timely, autonomous, comprehensive, extensible, non-intrusive, resilient, reliable, available, and accurate. A scalable monitoring system is able to handle a large and increasing number of probes without issue. This is important because the potential number of deployment instances can be very large and consume a lot of resources. A monitoring system is elastic if it can handle

monitoring very dynamic resources such as instances being created and destroyed. A probe is adaptable if it avoids hogging resources of the system it is monitoring when those resources may be needed by the monitored system to perform its task. A probe is timely if data arrives to the consumer in a time that is reasonable for intended purposes. A monitoring system is autonomous if it self-manages and distributes resources without challenge to users. Comprehensiveness implies that the probe supports many different resources and systems to be monitored. An extensible monitoring system allows the system to be extended to new functions. The system should not be intrusive, where intrusiveness would require modification to the system or resource being monitored. A monitoring system is resilient if the user can trust it to to continue even after component failures. A reliable system performs required functions under normal conditions for a period of time and an available system handles user requests when they are made. Finally, an accurate system provides monitoring data that is as close as possible to the real value. These requirements, although relatively abstract, provide a basis for requirements of a monitoring solution in a general sense and as thus are adopted as general requirements for a monitoring solution to be designed by this project.

2.1.2 Costradamus: A Cost-Tracing System for Cloud-based Software Services

The article about the monitoring solution: Costradamus [13] focuses on the cost of *serverless computing*. It begins by discussing the growth of *serverless computing* as an architecture to reduce costs for development and operations and the pay-per-use cost that has accompanied cloud computing services. Usage and pricing models for popular public cloud providers, although accurate, lacks specificity when it comes to usage statistics and thus cost and waste of resources. Costradamus is specifically interested in solving this by providing per-request cost-tracing for applications based in cloud services such that users can optimize their deployments using this improved cost-awareness. This is the motivation for this is ultimately lowering waste and improving profit margins.

Based on their motivations, the writers define Software Service Cost Model Metrics which they use and are also useful to consider for anyone considering similar work. These include the Marginal Request Cost which is calculating the cost of any invocation of a request. It can also be broken down into cost per capacity type. The Marginal Request Cost is calculated by the units of computation used multiplied by the cost of one of those units. This is adaptable to other monitoring models. Finally, they define three types of waste to consider:

1. Metering Amount Waste : describes the difference between metered and measured consumed amount. For example, 68MB of memory is metered but never used.

2. Provisioning Duration Waste (PDW): Time that could be used for computing invocations of requests that isn't used.

3. Provisioning Amount Waste : Similar to PDW, the difference between how much computing power was provided versus how much was actually metered as used.

All these are important metrics when designing a monitoring system focused on cost and waste. These definitions provide a more specific definition of waste than that provided in Section 1.1.1 by breaking down unnecessary provisioning of Virtual Machines into the different ways provisioning can be necessary and thus waste created.

The creators of Costradomus also state non-intrusiveness as a design goal, something that can be desired in all types of monitoring systems. They go on to explain how they designed their experiments and evaluations although those are not necessarily relevant to this project.

2.1.3 Modeling and Managing Deployment Costs of Microservice-Based Cloud Applications

The important discussion of this study is how the model calculates cost. In [14] the authors used a Graph-Based Cost model in their implementation. This involves many parts and describes different costs that can be modeled. The total deployment cost is the sum of operating each individual service in the application. The model sees service quality as constant, so the service scales for higher loads and costs more as a result, resulting in a linear function to define cost per request. The metrics for this include four cost factors : compute costs (payment for CPU time), per request payments, costs of input/output operations, costs of additional operations (eg. elastic IP addresses). While in theory costs should be linear per request, spinning up new Virtual Machines leads to jumps where the cost per request is higher and then levels out. The overall shape is linear, but those jumps lead to inefficiency. The graph model itself models service call graph with a weighted directed acyclic graph, where the graph represents the paths a request can take between microservices and the weights represents the likelihood of a request needing to use that path. The service cost model has functions that represent the total operations cost in a given time period. This sums the total cost of services and depend on the type of service. *Lambda-Backed Microservices*, a model represented by "a Microservice built on top of so-called 'serverless' cloud services"[14, Page 2] are independent of eachother and their cost is simply a sum. *Instance-Backed Microservices*, any microservice defined by the need "to carry out the actual computation,

and to implement autoscaling and elasticity, the service uses a pool of backing instances of varying size."[14, Page 2] Instance-Backed Microservices need a per-request load factor for each endpoint. The total cost is a sum of all other costs and using these the total deployment cost can be calculated. Using "inward-facing workload" the system can calculate how many requests are being made to the system from outside. This can then calculate from these outside requests the likely amount of load inside the system resulting from these calls. This load can be calculated from previous data or presented for different assumed load levels.

Although the scope of the project (CostHat) developed for this article is much more fine grain in its attempt to monitor cost by attempting to monitor the costs of specific request to an API, it does represent an attempt at calculating costs of services deployed in instances, albeit with a different focus. This is a useful reference point when considering the cost of running and monitoring Virtual Machine instances.

2.1.4 About Monitoring in a Service World

Monitoring is needed at many different levels of abstraction. These can range from high-level monitoring which provides information about the status of the virtual platform to low-level monitoring done by the provider to collect specific hardware, operating system, middleware, or network infrastructure information. This monitoring data, once collected can be used for a variety of uses. These include:

- Adaptation Actions: Adapting and reconfiguring the system or resource usage to provide the service or provide the service more efficiently.
- Flexibility Support: Allowing the service to support different variable of load, service, and service type.
- Awareness Support: Ensuring the system data is visualized to an extent that allows the human users to interact with it naturally.

Any solution to the monitoring problem should consider these use-cases when attempting to visualize monitoring data with a focus on cost and waste. There should also be a clear definition of not only what to monitor, but also how to monitor deployments. Certain data may be not be important to some users and its collection would affect the efficiency of the monitoring solution. This is part of a discussion of the efficiency and effectiveness of a monitoring solution and how to balance these two often mutually exclusive qualities. This is important because this balance is something that must be decided by any monitoring solution and this project will attempt to seek a solution to this problem.

2.2 Current Solutions

For reference, a small summary of some of the major cloud monitoring options has been provided. This covers the basic functionality of these solutions

2.2.1 Cost Monitoring Through Provider Dashboards

These solutions are examples of provider-specific monitoring solutions for Virtual Machines. These are ultimately the final authority in monitoring Virtual Machines because they provide the final cost to the user and their external monitoring should be much more accurate than any internal monitoring solution because of are the only tools

Azure Monitor

https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview-azure-monitor Microsoft Cloud Monitoring offers monitoring services for its cloud solutions. Through the Azure Portal users can view basic monitoring data from active virtual machines and services.

Amazon CloudWatch

https://aws.amazon.com/cloudwatch/ Amazon Cloudwatch monitors any Amazon Web Services resources and applications. These provide real time metrics and allow user-defined rules for alarms based on this data. The user can also choose which metrics they wish to see.

Google StackDriver Monitoring

https://cloud.google.com/stackdriver/ Google StackDriver Monitoring offers monitoring for Google Cloud and Amazon Web Services deployments which is a multi-cloud feature not common on proprietary monitoring services.

2.2.2 Cost Monitoring Through Third-Party Applications

AppNeta

https://www.appneta.com/ AppNeta offers integrated monitoring solutions for private, public, and hybrid clouds as well as many other things. It is however unfortunately proprietary and its use and implementation not publicly available, making it not an alternative to what's trying to be achieved by this project.

2.2.3 Cost Monitoring Solutions Through Additional Instrumentation

NAGIOS

https://www.nagios.com NAGIOS is a current solutions for monitoring. It monitors applications, services, operating systems, network protocols, system metrics, and infrastructure components all within one tool. All this information is provided to a singular centralized web interface in order to best view the status of the things being monitored. Although NAGIOS is an interesting and well-fleshed-out solution, it is not quite as lightweight and does not have the focus on cost and waste that a potential solution to this problem requires.

NetData

https://github.com/firehol/netdata This is an open-source solution. It works by hosting the monitoring service on the instance being probed. In this way probe data is collected by physically viewing the site. While objectively successful at what it aims to do, this tool does not provide the lightweightness desired for a monitoring solution or the focus on cost and waste.

Grafana

https://grafana.com/ Grafana provides data visualization and aggregation for monitoring data. This data is provided by the user via the method of their choosing. This is an interesting dashboard example, although not in-line with the requirements for this project.

Chapter 3

Requirements and Design

3.1 Requirements

In addition to the general attributes of a monitoring solution which can be drawn from the literature, particularly in Section 2.1.1, specific requirements can be elicited based on what is already understood as a high-level solution, the related works, and the scope of the project. These requirements can be split into requirements for a probe, representing a solution for collecting data from monitored instances, a need for which is established in Section 1.2, and requirements for an aggregator, representing the need to process collected monitoring data into human-readable representations, the need for which is established in the same Section 1.2. These two parts are not necessarily distinct, although they can be. Other requirements not related to these two parts can also be elicited separately, particularly if they are more abstract.

3.1.1 Probe Requirements

- The probe should be universally compatible within certain restrictions. The probe should be functional on all common cloud providers and should at least be deployable to Linux Virtual Machine types on these public clouds.
- A probe is associated with a certain *deployment* - a deployment being any software that has been deployed to a cloud provider for hosting purposes. A user can have many different applications or parts of an application deployed to cloud-based *Virtual Machines* - a computer that runs an operating system, but entirely in software. It is key that users can probe these separately and the data is properly associated on the

aggregator - the application that takes probe data and processes it - with a specific application deployment.

- A probe in an instance of a deployment can be differentiated from another instance of the same deployment. This is essential because different instances of an application deployment can have different usages. One of the key aspects of the project is to ensure that a user can properly cost-trace their deployment. If a certain instance is contributing to waste that is something that the user should be aware of.

- Probe data should be considered sensitive and secured accordingly. Although the intention is that the software providing this cost-monitoring, waste-calculating service is open source and free to use, the applications being monitored might be proprietary and malicious users should not be able to intercept and use data being sent from the probe to the aggregator. Since interception can not be guaranteed to be avoided, *encryption* - the process of converting information or data into a code - should be used to ensure the data can't be used.

- The probe should be configurable, but easy to use. When a probe is embedded in an application instance, although configuration is required in order to accomplish other requirements, it is necessary that the probe be easy and quick to setup in order to ensure monitoring is available to the largest common denominator of users. Part of this requirement also includes the requirement to be non-intrusive. This is somewhat of a subjective requirement, but should be considered nonetheless.

- The probe should strike a balance between efficiency and effectiveness. The probe itself uses part of the Virtual Machine and is, in a sense, itself waste; should the rest of the application be perfectly optimized already. As such, a decision should be made on how much computational power and/or network capacity the probe should use while still ensuring the data is effective and usable. This can also possibly be configurable.

- Critically, the probe should collect utilization statistics on the monitored system. The statistics collected can vary in nature, but should include some of either data metrics such as: CPU speed, CPU utilization, disk throughput, system up-time, memory utilization, or network based metrics such as jitter, packet loss, traffic volume, etc. [1] [7] [22] [11] [23] [3].

3.1.2 Aggregator Requirements

Although a specific *network topology* - the arrangement of the elements of a communication network. - has not yet been decided, the way a potential user interacts with the service will involve data being served to the client in someway or another. As such requirements for such a service can already be elicited:

- The aggregator should provide some visualization tools for cost and waste, itself. One of the key parts of the system is its ability to visualize cost and waste for a potential user. As such multiple visualization options with different foci should be designed.
- There should be some kind of authentication system in order to ensure integrity of data and separation of user data. Ideally this authentication system should be easy-to-use and non-intrusive, avoiding barrier-to-entry.
- All data should be communicated with a protocol that is reliable enough to maintain a considerable level of data integrity.
- The aggregator should store probe data so it can be used later, but also such that it is separated by user to avoid data being accessed by users who don't have proper permissions.

3.1.3 Other Requirements

- The system should be open-source and extensible.

3.2 Design

3.2.1 Network Topology

Considering the multitude of requirements for the desired system a network topology can be considered. A network topology is "The specific physical, i.e., real, or logical, i.e., virtual, arrangement of the elements of a network." [5]

Point-to-Point

Also known as peer-to-peer, this network topology is favoured for this maximally distributed nature. A potential application of this network topology would likely see probes communicating between each other. It could also be made such that no central server service would be required and as such reduce development load and cost to the service. This is a sizable

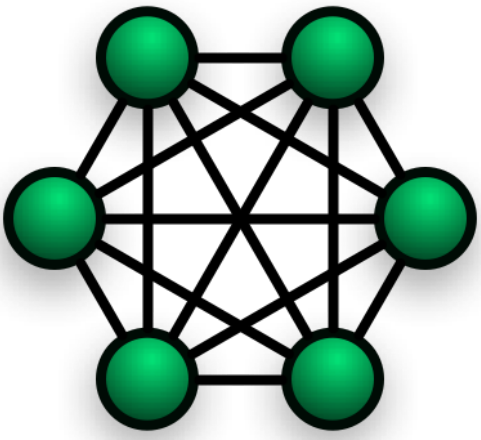

Fig. 3.1 A basic graphic of a mesh topology

Source : Wikipedia[25]

advantage for this network topology. Unfortunately, there are also considerable disadvantages to this choice, most of which are issues with the implementation of cloud services that make such a topology difficult, rather than issues with the topology itself in theory. The abstraction provided by cloud services means that it becomes extremely difficult to communicate, not just between Virtual Machines on the cloud service, but also Virtual Machines across cloud services. This might be an obstacle that's possible to overcome given enough effort, but issues might arise if it were desired to eventually expand the system to work with something like Docker containers or just issues with firewalls interfering with outward communications. The second downside to a peer to peer network is that with all the computation and storage happening in the probe the efficiency of the probing system itself drops considerably. A lot of computational cost is placed on the user's application. Considering the final objective of such a monitoring system is to reduce cost and waste, adding a considerable amount in order to achieve that seems counterproductive. Any topology of this type would be envisioned as a partially connected mesh topology where each node only is connected to $\log_2(N)$ other nodes to the end that all Nodes are connected through each other.

Mesh

A mesh version of the network would operate very similarly to the peer to peer system and could be even considered a variation of such, except with every peer knowing every other peer directly. Such a network can be disqualified on the fact that it would reduce efficiency

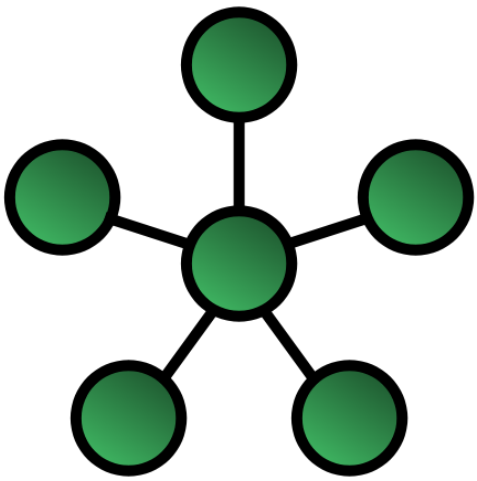

Fig. 3.2 A basic graphic of a star topology

Source : Wikipedia[26]

even more because of its poor scaling. The number of connections in such a topology would be $sum_N = \frac{N(N-1)}{2}$. Its very clear that with scaling sum_N becomes very large very quickly and waste would become very large. The failsafe abilities of such a network are not worth the cost for the application in question. Other topologies similar to mesh include Daisy Chain and Ring. These don't have the issue of scalability as a standard mesh, but also have very limited failsafe ability and no advantage for the system in question and as such are not considered. A basic mesh topology is visualized in Figure 3.1.

Star

A star network in this system would involve all probes reporting to a central server. This is in terms of complexity among the least complex options. By reporting data to a central server, most of the work is placed on this central server which provides the data aggregation service and probes are therefore able to be much more efficient. It also provides a central address for potential users to access data. The obvious downside to such a design is the centralized nature itself. Should the central server fail in the best case scenario there's significant downtime. This stops not just users from using the data, but also stops probes from reporting their data. This could cause lots of data to build-up in the probes and possibly cost to the users' application deployments. As well, when the probes finally can report to the central server the high usage to transmit so much data could cause another failure. As such, a topology with a single point of failure should be avoided. A basic star topology is visualized in Figure 3.2.

Hub and Spoke

A hub and spoke topology expands on the star topology by essentially horizontally scaling the central server and providing redundancy. Many servers as instances of the central server handle probe post requests. Although this design is not entirely failsafe, it is a significant improvement over strict star design for not just its failsafe ability, but also scalability. The hub can be scaled horizontally to add more servers as needed to meet load. Although, scalability is not a central concern for the system in question, a topology that allows for easy scalability is always a plus. The minimal negative attributes and considerable amount of advantages makes hub and spoke the optimal choice for this system.

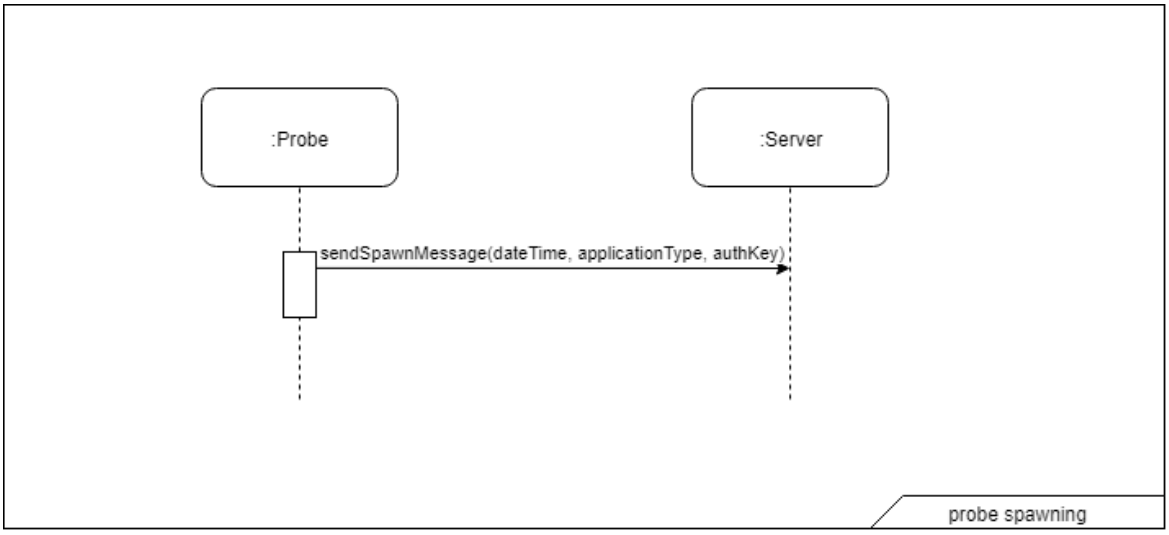

Fig. 3.3 Probe sends message to the server to notify server of it's existence

3.2.2 Protocol Design Considerations

Considering the network topologies discussed in Section 3.2.1 the hub-and-spoke topology becomes the optimal choice for the aggregator probe relationship because it provides the low probe-side *overhead* - cost or expense - required for low waste as well as the simplicity of expansion to future non-Virtual Machine-specific applications and acceptable level of reliability necessary for the project-at-hand.

Data Association

The most apparent issue when it comes to associating data with the correct deployment is the issue of multiple probes reporting data for different instances of the same deployment. This

situation would be by design, but would require careful consideration to differentiate probes. A solution shown in Figure 3.3 uses a dateTime (at parameter representing a specific date and time), authKey (a parameter representing a key used for authentication) combination in order to uniquely identify a probe. This combination can be sent in any data report. There is of course the possibility of identification collisions if probes use the same dateTime. The dateTime can be made accurate to a level (such as nanoseconds) that collisions become extremely unlikely if collisions become too common.

Data Security

In order to ensure integrity of data the probe data will need to be encrypted as interception is always a possibility in any network. A simple solution to this problem is embedding a public key in every probe with the intention of using RSA encryption. The matching private key to this public key will be stored by the aggregator and kept secret. Access to this private key should be regulated through the use of a vault software to ensure it is never exposed to malicious users. It is possible to have public/private key pairs for each application, but it is probably unnecessary as there's no reason for the private key to ever be exposed to any of the users intentionally. A solution to this issue already exists in the form of HTTPS. This protocol uses standardized protocols to distribute and encrypt packets using RSA encryption and as such would make an ideal candidate for security.

Push vs Pull

There exists two possibilities for how to transfer the data from the probe (the data creator) to the aggregator (the data consumer). One is the pull method: where the data consumer requests that the data creators send their data when it the data consumer wants it. The second is the push method: where data creators send data to the consumers when they are ready to do so. Both of these methods have their strengths and weaknesses and even a combination of them is possible. The pull method allows the aggregator to pull data when it needs it and thus ensure that data is not sent when it does not have the resources available to handle that data. The push method does not have this advantage, but it does have a significant advantage that isn't necessarily related to efficiency. The push method means that the data creator needs to know the location on the network of the data consumer, but without a hybrid approach, the reverse is not necessary. This is key when there are a potentially very large number of probes on the network, being created, being deleted, effectively moving in some scenarios. This makes keeping track of the probes a difficult prospect for the aggregator should it want to know the positions of the creators. On the other hand, the aggregator will

ideally always be located in the position it specified to the probes at the time of their creation. This makes the push method more ideal. As well, the use of only the push method and not some form of a hybrid push/pull method means that the probes can be extended to even more virtualized deployments (eg. docker containers) without issue in getting the data delivered to the consumer. As such, only a data push will be implemented for this design.

Reliability

Reliability becomes an issue when a lot of data is being transfered, and especially when the life if the data producer is relatively fleeting. Although it could be argued that some lost application monitoring data is acceptable because the data is non-critical any lost data degrades the overall integrity of the remaining data. As such, data loss should be avoided and reliability of probe data arriving at the aggregator considered a priority. For protocol considerations to this end, research in Internet of Things draws many parallels. The discussion for Internet of Things protocols involves balancing tradeoffs in reliability and resource constraints. A paper on reliability in the Internet of Things [20] evaluates two protocols and concludes that while one protocol is more reliable and more costly, whereas the other protocol is less reliable, but also less costly, ultimately the choice of protocol comes down to the importance of reliability vs cost. This same consideration must be taken in this project. The increased latency of the more reliable solution does not affect this project significantly and as such for the purposes of this design reliability trumps cost, as probes are embedded in powerful-enough Virtual Machines, but an evaluation of the probe's inherent waste might indicate an alternative choice is necessary for future implementations.

3.2.3 Probe Design

Any design for the probe must first take into consideration the requirements for the probe elicited in Section 3.1.1. In order to deal with the probe uniqueness problem two methods are considered. Firstly, a probe will be associated with an application via a key. This key is called `authKey` in Figure 3.3. This allows a probe's data to only be associated with a specific application deployment when the data is aggregated by the aggregator. This solution is part of the probe's design. It is not sufficient to differentiate probes on its own, but the remainder is solved in the application's protocol.

In order to guarantee an optimal balance between efficiency and effectiveness a configuration for dataInterval will be provided when the user creates the probe script as seen in Figure 3.4. The idea behind this value is that it will set the interval at which the probe gathers data from the system. `POST` data from the probe to the aggregator will have its time interval

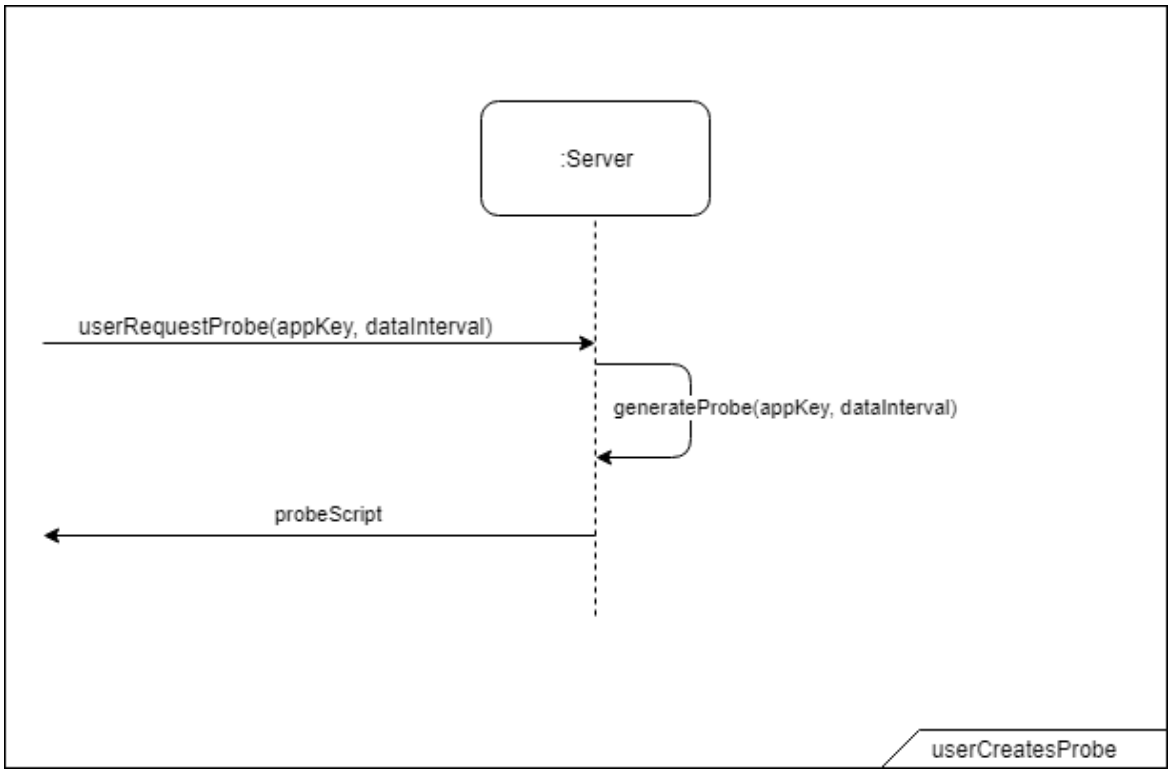

Fig. 3.4 User requests a probe script from the server

adjusted according to this dataInterval value. This way the user decides the balance between effective and efficient that suits their needs.

In order to fulfill and extend non-intrusiveness to the user the probe will be configured to allow it to run without user intervention and continuously. This means that a user will have the option to place it as a startup application using the method of their choice and have that probe start when the image it is embedded in also starts. This makes the probe easy-to-use.

3.2.4 Aggregator Design

The aggregator's protocol is in fact the hub-and-spoke topology as per the designed protocol. However, when serving data to users who request it the aggregator takes on more of a server-client relationship. Therefore, it can be considered that the aggregator is also a server. This server has requirements not just for dealing with probes, which is mostly considered part of the protocol design, but also for what kind of information it provides to clients.

One of the server's main requirements is that it visualize data for users. In order to visualize the data the aggregator is provided by the probes in a way that represents waste and cost over time it is probably best to graph the data. This would be done on user request as seen in Figure 3.5.

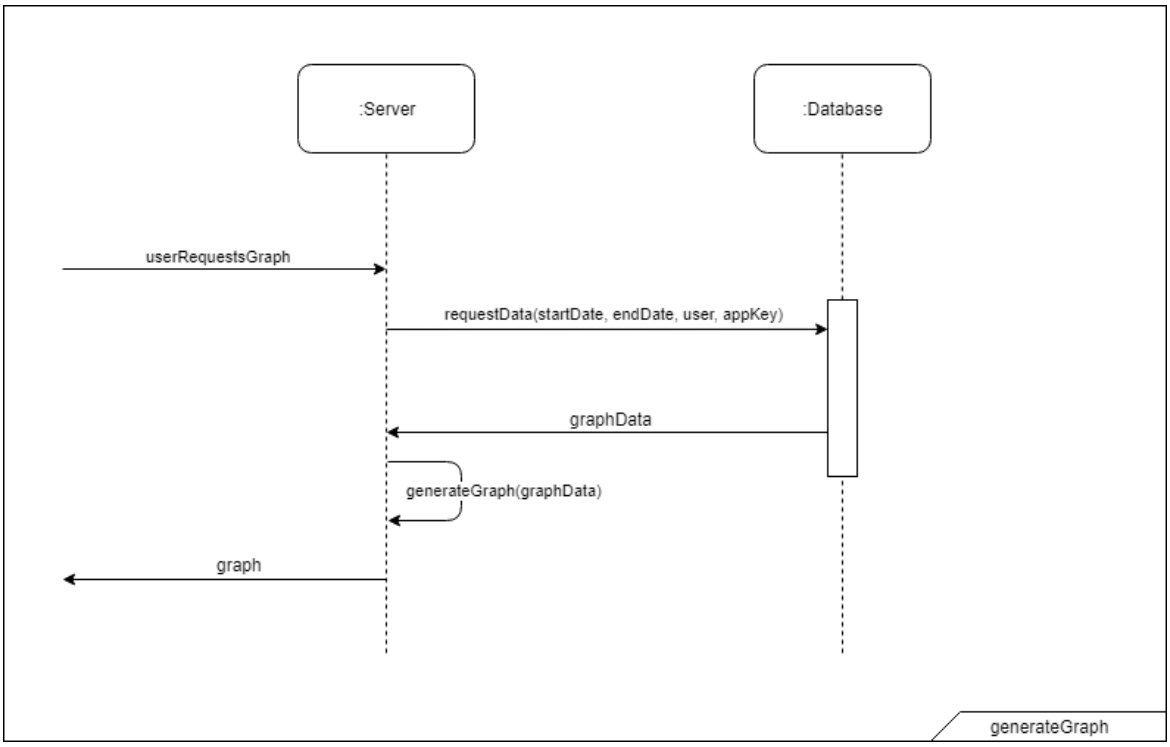

Fig. 3.5 User requests graph, aggregator generates graph

Multi-tenancy

Multi-tenancy "refers to a software architecture in which a single instance of software runs on a server and serves multiple tenants"[12], whereas a *tenant* "is a group of users who share a common access with specific privileges to the software instance" [12]. Multi-tenancy will be a necessary facet in order to ensure integrity of user-data within the database. All applications and data will be stored in this database and an overlay of the database layout can be represented by Figure 3.6.

As any user will need to be safely authenticated, authentication for the aggregator will be done through a third-party authentication provider to limit local security issues. The token and associated user data will be stored in a aggregator-side database as well as the user's browser session in order to facilitate easy-use.

3.2.5 API Design Considerations

An *API* could be a useful part component to the monitoring solution. Although not strictly critical to the monitoring itself, it could provide valuable access to users in order to access the collected data for further processing or private storage. A well-implemented API could be integrated into the solution for delivering visualized data in consumable chunks if such an

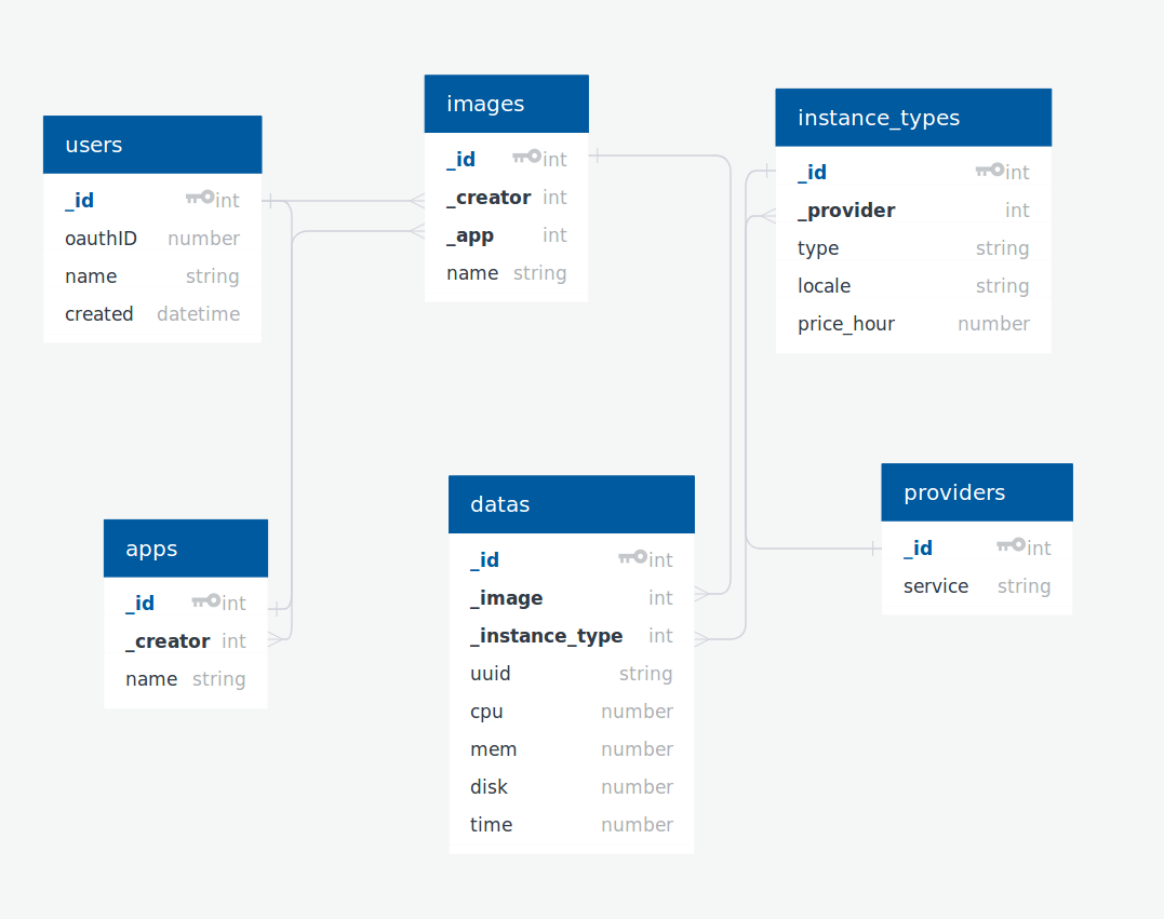

Fig. 3.6 A basic design for a database.

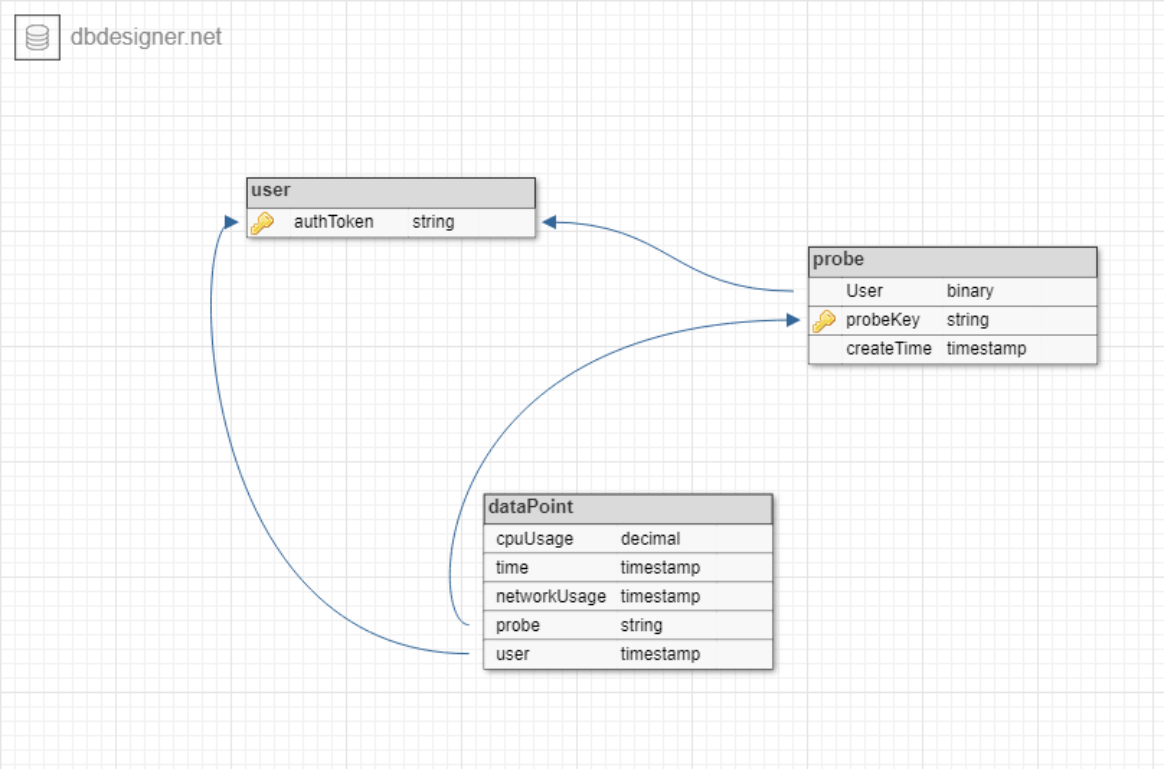

Fig. 3.7 The process of a post request to the API.

optimization were to be implemented. A process of a post request to an API can be seen in Figure 3.7 and would operate, or by synonymous to, depending on the implementation of the service itself, the methodology for processing data for visualization.

3.2.6 Architecture

All subsystems considered, the overall system would operate like Figure 3.8. In this it can be seen that the probe is embedded in the Operating System with whatever application is being monitored. This Operating System is contained within the Virtual Machine. The Probe has local storage which it can use in case of connection failure. If connection exists, the probe can communicate the data it collects to the Server which then stores or retrieves that data from the database.

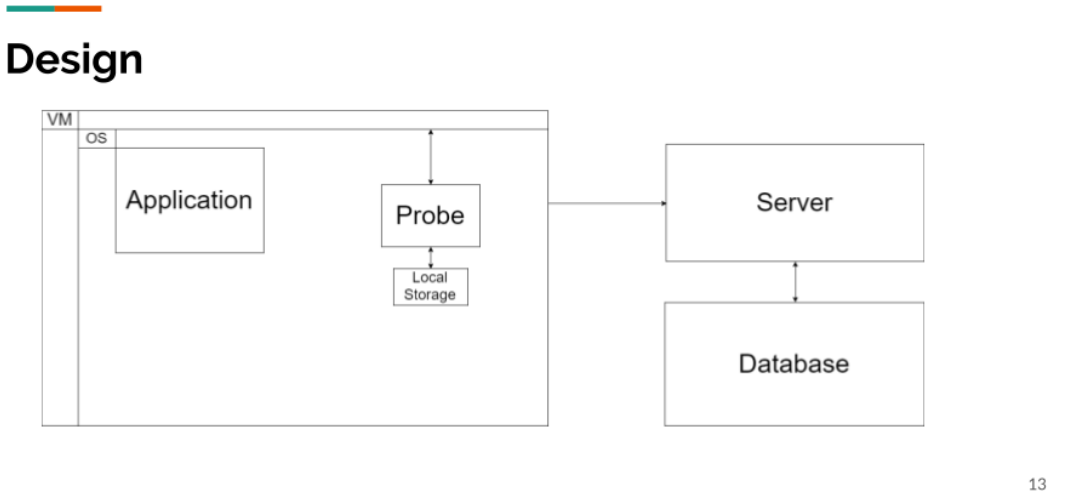

Fig. 3.8 A preliminary and basic design for a monitoring solution as a service.

Chapter 4

Implementation and Evaluation

4.1 Implementation

A code repository for this implementation is located at https://github.com/a-d-spina-student/waste-cloud-computing. This repository contains information on how to use the tool as well as

4.1.1 Probe

In Section 3.1.1 it was decided to make the probe as universally compatible as possible. As such, only simple bash commands such as `cat/` `grep/` `sed/` `curl` were used in the hope that the majority of VMs running Linux Operating Systems would be able to run the probe script. The probe script itself simply collects the data from the system and sends it via `curl` to the aggregator. It also sends useful tokens to the aggregator to help segregate data points.

An important feature of the probe is its ability to store data locally in a file if for some reason a connection fails to establish with the service. It will continue to store these data points and retry sending them as long as the network fails. In this way, when the network is established again the aggregator will have up to date information about the instance being probed. The probe gathers data as seen in Listing 4.1. The full code can be seen in the appendix in Section 5.2.

Listing 4.1 An example of a generated bash probe

```
#!/bin/bash

#Gather new data
        memFree=$(awk '/MemFree/ {printf( "%f\n", $2)}'
    /proc/meminfo)
    memTotal=$(awk '/MemTotal/ {printf( "%f\n", $2 )}'
    /proc/meminfo)
    MEMORY=$(free -m | awk 'NR==2{printf "%.2f",$3*100/$2 }')
    #Shows memory usage without buff/cache included
    diskSize=$(df --output=size -B 1 "$PWD" |tail -n 1)
    diskUsed=$(df --output=used -B 1 "$PWD" |tail -n 1)
        DISK=$(awk "BEGIN {printf \"%.2f
        \",${diskUsed}/${diskSize}*100}")
    CPU_usage
    CPU=$?
    UUID=$(dmidecode | grep -i uuid | awk '{print $2}' |
    tr '[:upper:]' '[:lower:]')
    TIME=$(date +%s)
```

4.1.2 Aggregator

Technologies

The aggregator was made using simply NodeJS (v10.8.0) with Express (v4.16.3) and some extra libraries such as PassportJS (v0.4.0) and Mongoose (v5.2.6). The aggregator communicates with a MongoDB (v3.1.1) database.

Service

An example of the project hosted at https://www.universalcloudmonitoring.com/. From the homepage shown in Figure 4.1, through the menu, users can access the View Profile page for profile management where they can add applications and images of those applications they want monitored. A probe script can be generated based upon these parameters. Also, users can view data they've collected by accessing the View Data page. Users can also login and accessing protected pages requires a login.

Fig. 4.1 The homepage of the service.

Fig. 4.2 An example of probe data displayed to a user. One plot is zoomed for added detail.

Visualization

Visualization of data is accomplished via the Plotly (v1.0.6). Figure 4.2 shows the visualizations available. Visualizations include a visualization over time of Virtual Machine utilization and a breakdown of cost and waste for each Virtual Machine. These are grouped by image and then by application.

Data Aggregation

When the aggregator, which is implemented as a server, receives a data point from a probe it simply stores it in the database of data points. All data is stored in a cloud-based database which the service maintains a connection to. This database can be substituted for a locally-based one or any other cloud-based database. This provides greater flexibility for users seeking to deploy the created aggregator service.

When a user requests information on certain images they have registered, the aggregated queries the relevant data points and runs and algorithm to determine the information needed for the user. The algorithm runs in the following way and expressed visually in Figure 4.3:

1. Sort queried data points by time.

2. Add data points to buckets based on their associated image.

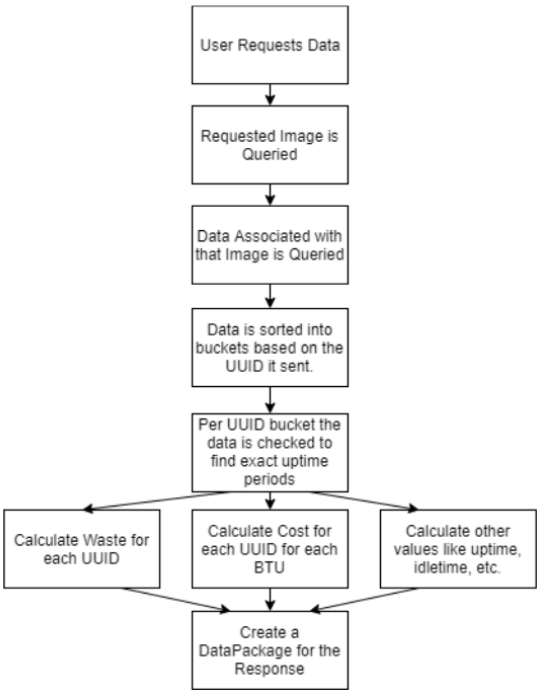

Fig. 4.3 A visualization of the algorithm for processing monitoring data

3. For each image bucket, sort data points into buckets based on their Universally Unique Identifier (UUID).
4. For each UUID bucket
   (a) Move through the data points that are sorted by time and keep a running average of response time
   (b) When there is a gap in response time from the probe that exceeds a set multiple of the average response time, consider that period a dead period and calculate if the dead period ended a billing time unit, calculating cost accordingly.
   (c) Start a new billing time unit if the previous one was ended.
5. Calculate all metrics on a per instance basis and send sorted data package to the user.

In this way data points are processed in a linear way and served to the user. Another important algorithm to discuss is the way cost is calculated. For the purpose of this system Billing Time Units (BTUs) are set at periods of 60 minutes. This is a limitation of the system as BTUs can vary in length, but the dataset being used for known provider costs does not specify them. When the previous algorithm is processing per-instance data, the cost is calculated by finding these Billing Time Units and adding to the total cost for them. A BTU starts when the first

Fig. 4.4 How to manage a user's profile in the service

data point is received. No cost will be calculated unless it is found that a probe has stopped reporting. This can also be the last probe ping. The time since the most recent probe death is divided by billing time units and the total cost is increased. If there is a period in that billing time unit for which the probe indicated the instance was dead, but because of the size of a unit the cost was larger than the specific uptime would suggest, this is considered waste and is added as such. As well, if the CPU usage is below some threshold of utilization, for example just system CPU utilization as the majority, this is considered wasted time as well and the cost of this time will be added to the waste calculation. All this processed data is packaged into a data package and sent to the client.

Profile Management

Users who sign in via third party authentication with Github have data saved for their monitored applications. The profile management page of the service can be seen in Figure 4.4. They can register new applications they want to monitor as well as new images they have saved that are part of that application that they want to monitor. In this way they can monitor multiple parts of the same applications separately. As well, once an image is created, a probe script can be generated for it. The user selects the type of instance that is running and then the service finds the data for that instance and inserts it into the final script. In this way accurate cost reporting can be achieved.

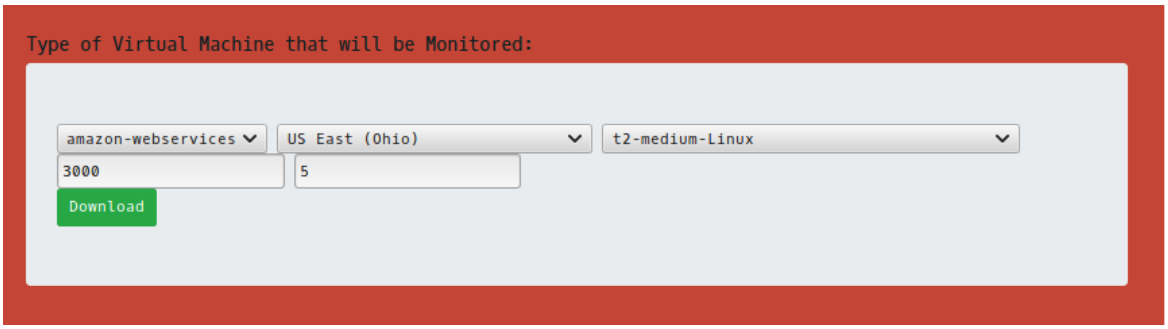

Fig. 4.5 Generating a probe for an image

As well, users can select the ping rate of their probe. Higher ping rates will mean more accurate data and thus effective data while lower ping rates will mean more efficient but less accurate data collection.

4.2 Evaluation

The requirements of a monitoring system that were elicited in Chapter 2 were both functional and non-functional. This means that some requirements were of the functions that a monitoring system would have to perform to meet the specifications of the the project and some requirements were qualitative of the system itself.

4.2.1 Functional Requirements

Probe

Probes are associated with different deployments by allowing users to create images in their profile management. These images represent versions of deployments that will be monitored. By creating an image, the user is specifying that any probe generated for that image will report that it is of that deployment. This obviously leaves room for user error should users not update images when new versions of the deployment are created, but the system is functional.

Probes generated for the same image and instance type identical in implementation. In order to avoid multiple instances of the same image/deployment returning combined data a UUID is used to distinguish one instance from another. Major cloud providers guarantee that UUIDs of their Virtual Machines they host will be unique, but this uniqueness is not guaranteed across multiple platforms. As such, universal and generalized monitoring is possible, but the possibility of UUID collision exists across a hybrid cloud. The low collision

rate on UUIDs makes this collision extremely unlikely, but it can not be guaranteed that the VMs would be uniquely identified in this use case.

Probe data is secured using RSA encryption because the service uses HTTPS. Any curl requests will first be encrypted with the service's public key before sending and any data viewed by the user will do the same. In this way data is secured. If a user wanted to host the service for their own uses, a connection through HTTPS would be necessary in order to guarantee privacy and security.

Probes were made configurable by allowing the ping rate/instance type to be set by the user. They are simple to use because the service generates the probes for the user. As such, the user simply has to follow a set of instructions to place and activate the probe within a VM image.

Because probes are configurable in their ping rate, the balance between efficiency/effectiveness is effectively limited by user choice alone. There is some limit in terms of network/service capacity in terms of the level of effectiveness that can be achieved, but should the user need to shrink the load of the probe itself the ping rate can be increased to a theoretically infinite level to lower monitoring load, although the drop off in effectiveness would match the increase in efficiency. With such a balance available, the probe is in effect both effective and efficient.

The probe collects data metrics from the monitored system. Currently these metrics include: system up-time, disk utilization, memory usage, and cpu utilization. Although this is a satisfactory beginning, this could easily be extended to allow for more metrics to be collected, particularly network utilization.

Aggregator

Aggregator visualization was achieved through the use of plots. These plot types were chosen with the intention of providing the most effective visualization of the collected data. A stacked bar plot represents the total cost accrued in billing time units. This allows not only cost in a billing time unit to be represented, but also the ability to stack the cost from different instance types of the same image to show how different Virtual Machines had different cost contributions. A line plot represents the utilization statistics. This is an ideal choice because the data points consist of values over time and a line plot effectively represents the relationship to time. Finally, a donut chart represents total cost and waste effected. This allows an effective representation of proportions when relating the overall value of the wasted compute power in relation to the used power.

The aggregator does provide an authentication system. It is secure because it uses Github third-party authentication which provides a trusted authentication system and a token when

logged-in. This is easy-to-use because it simply requires the click of a button in most cases because Github's cookies will see that the user is logged in with Github already. Should the user not have this cookie, the login process is a simple username and password. It is non-intrusive and has a low barrier-to-entry for the same reasons. This authentication system also provides extensibility for the project if users were to be made able to link Github projects to the monitoring system.

The protocol for data aggregation is described in Section 3.2.2. Through this protocol, even network or service failures can be mitigated because data points are timestamped. This means that regardless of the time of arrival of any data packet the data will maintain its integrity. The only point of failure is if the instance which hosts the probe fails during the network or service failure. This is a known issue and only mitigatable through improved infrastructure which is beyond the scope of the service. As well, probe data is secure because the HTTPS is used to secure the transfer of probe data and visualization data. HTTPS uses widely trusted RSA encryption and key-exchange protocols that is the industry-standard for security.

The aggregator stores probe data in a database. This database is separate from the service itself. Because the service was implemented with a cloud-based database users are able to host their own service with their own database or theoretically host separate service instances connected to the same database and allow for data continuity. Users are given an id which is a token and only access data for their user id.

4.2.2 Non-Functional Requirements

Scalable, Elastic, Extensible, and Timely

The service designed by this project has an ability to scale in some ways. The scalability comes from the form of horizontal scaling. The service can be hosted on many machines and the traffic can be directed to the different instances in order to reduce load on individual machines. This means that the theoretical limit to user and probe traffic for the service is quite high and not of concern. The issue in scalability comes from number of data points submitted by probes. Due to multi-tenancy, users can access at most the data points associated with images that they own. This limit is too high because the number of data points for a user can be very large and grows rapidly with the number of probes deployed, especially high resolution probes. Although this problem can be partially solved through the use of vertical scaling in order to process the data quicker, optimization is needed in order to provide a timely and consistent user experience. Obvious optimizations could include limiting the time range and number of images a user can view or query at one time and also possibly caching

already processed data results in order to avoid repeat work the next time the user requests a data set. These optimizations are very apparent candidates for future work.

The service is elastic because of its ability to monitor a varying amount of unique instances. This is because instances can be started, stopped, or terminated, but their Universally Unique Identifier (UUID) will be maintained. These UUIDs can be used to identify different Virtual Machines as distinct from one another. The probes send the UUID of the instance they are monitoring to the service. Most major cloud providers generate UUIDs in a way that allows not only identification of which Cloud Provider is hosting the Virtual Machine, should that information be desired, but also the avoidance of UUID collision for the foreseeable future of cloud computing.

A large part of development decisions in the design of the service and particularly the probe is the need for extensibility. This is why probes only push data to the service, but the service never attempts to locate or pull data from probes because other future cloud services might not necessarily provide the web location of computing spaces. As well, probes send data in Javascript Object Notation (JSON) which allows for dynamically sized data packets and near-universal compatibility in the web development sphere. This would allow for future compatibility with an API or other types of data such as network data to be added with little modification to the service.

The service has the ability to be timely. Timeliness relies on the service providing the results to users when the users need them. In this respect data is always available to users when the service is live. This issues with scalability already established mean that timeliness could be affected by high load. These same arguments can be applied to the need for the service to be available. Should availability and timeliness be compromised, the service can be made more reliable and resilient by scaling the number of hosting instances horizontally such that many instances are available in case of one's failure and the integrity of the service is maintained.

Autonomous and Adaptable

The probing solution is autonomous because it does not require manual intervention to adapt to changing scenarios. This is particularly true in the case where the connection to the aggregator service fails. In this scenario the probe saves the data it has collected to be used later should it be possible to reestablish a connection. It can be seen that this solution to autonomy does not severely affect the performance of the the instance being probed, even over along period of failure in Figure 4.6. Even after an hour of service failure, the probed instance is only consuming less than half a percent more CPU utilization than it would under normal operating conditions. It can be argued then, that even in a state of service failure,

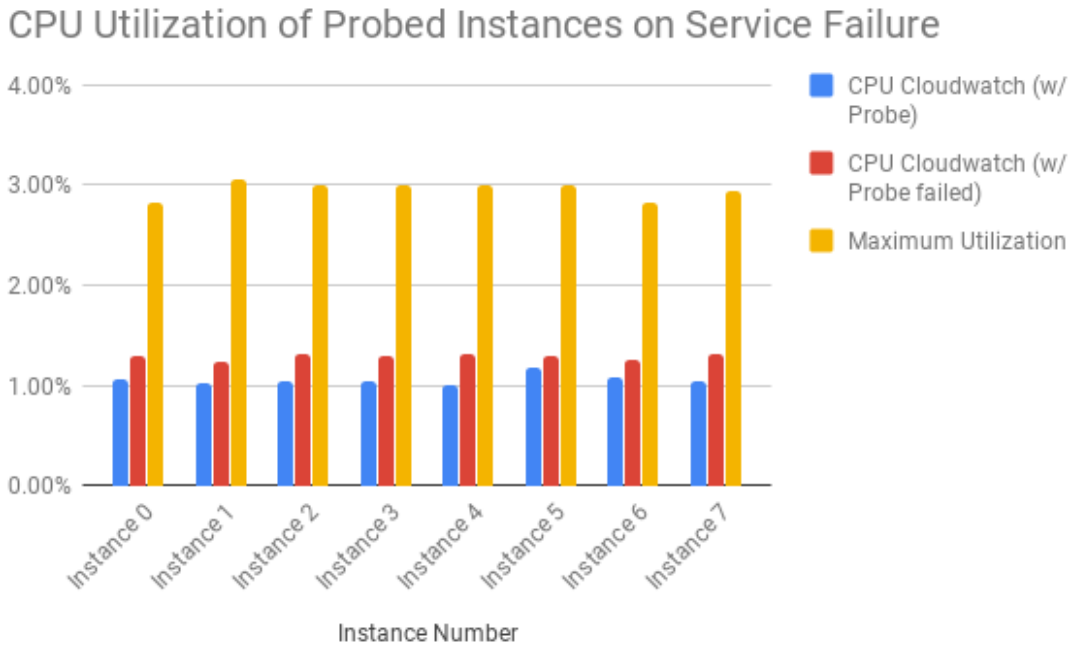

Fig. 4.6 Data showing the utilization of a Virtual Machine from left to right in the bars: Utilization reported by Amazon Cloudwatch under normal conditions, utilization reported by Amazon Cloudwatch during service failure, utilization reported by Amazon Cloudwatch on reconnection to the service after an hour.

the probe is still relatively lightweight. Even at the critical moment when connection to the service is reestablished the maximum CPU utilization of any tested instance is 3.05%. Although this is a rather meager sum, the test was only after 1 hour of failure. Should the service failure last longer, it can be reasonably expected that the CPU utilization on service connection reestablishment would grow roughly linearly as the amount of data to be transferred at that moment would also grow linearly. This can be considered a failing of the system and an optimal implementation might experiment with regulated transfer to maintain non-intrusiveness of the probe even in these scenarios.

In order to satisfy the need for a monitoring solution to be adaptable the probe was designed to be configurable. There are, of course, many other was a solution can be adaptable and this solution may not satisfy those, for example: it might be desired that the solution does not submit monitoring data until CPU utilization on the monitored system drops below a certain utilization threshold, but that was not included in this solution. Instead, the solution adapts to the changing needs of a customer and dynamic capabilities of Virtual Machines by being configurable. Users can choose a theoretically infinite variety of resolutions of

monitoring in order to adapt to their needs. The configurability can be seen in the appendix as code at Section 5.2 and the service at Figure 4.5.

These fields allow the use to configure their ping rate of the probe in order to provide different resolutions.

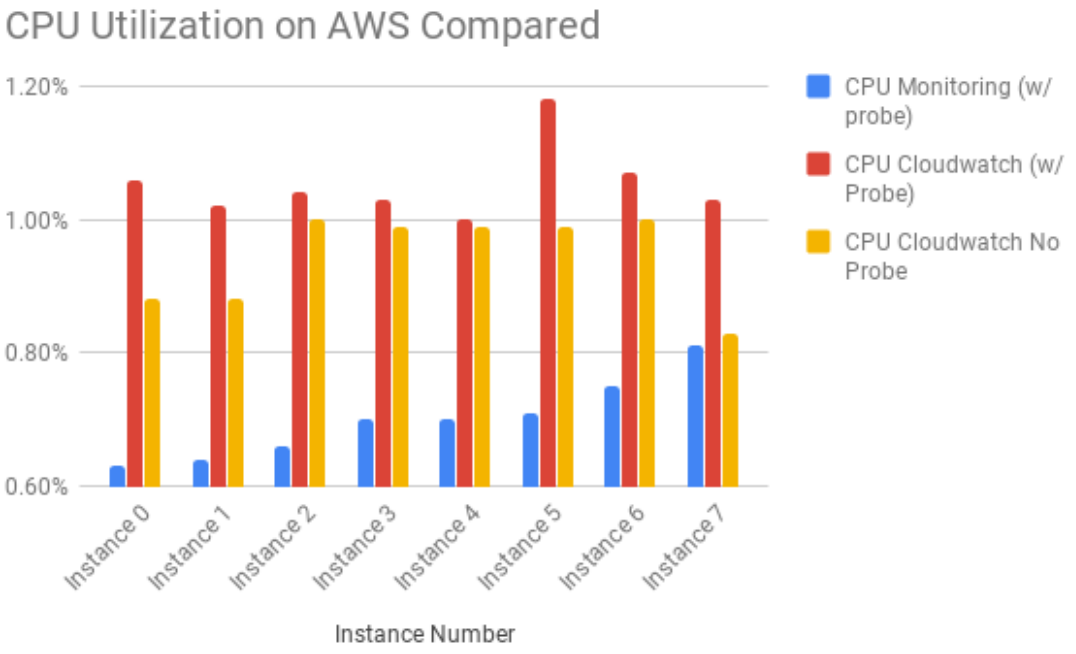

Fig. 4.7 Data for probe CPU usage on AWS Cloudwatch compared to CPU levels reported by monitoring solution.

Non-intrusive and Comprehensive

The probe is designed in a way that it is lightweight. This level of low-intrusiveness can be seen in Figure 4.7. The probed instances with and without probes register very low difference in average CPU usage over long periods of utilization. The difference peaks at 0.18% different in utilization in Instance 0. Even this is relatively low considering the small fraction of the total compute power available that is being used by the probing solution and the fact that this test was conducted on a t2-micro-linux provided by Amazon Web Services EC2 which is a relatively very low power Virtual Machine. The monitoring solution is also non-intrusive because it avoids the need for uncommon dependencies. The probing solution uses commands that are commonly available in common Linux operating systems to ensure compatibility with as many Virtual Machine types as possible.

The solution to the monitoring problem is comprehensive because of the generalization of the way usage statistics are gathered and the way the probe is installed. The probe simply has to be placed inside the image that will be run as a Virtual Machine. The user indicates that the probe should be started either as a service or as part of a crontab. The probe then handles itself by reporting data and then sleeping for the appropriate amount of time so as to be non-intrusive. This means that the probing solution is viable for most Linux-based Virtual Machines as cron and services are standard on these. This also means that the system will work on any cloud provider that hosts Linux-based systems and even potentially personal computers with Linux operating systems, although UUID collisions are not guaranteed to be avoided in this scenario. Three systems being probed simultaneously on different cloud providers can be seen in Figure 4.8

Accurate

Accuracy is somewhat of a point of weakness of the monitoring solution. The results returned by the probe are accurate to the hardware that hosts the Virtual Machines. This is because the data is gleamed directly from the top command of the Linux operating system. The discrepancy in accuracy arises from the external reported usage provided by cloud hosts' data and the probed data when the Virtual Machines are hosted on shared hardware. Virtual Machines on shared hardware only have a share of the compute time on that hardware. This means that the remaining share is used by other users. This does not present an issue to the users of the Virtual Machines, but it does mean that the utilization statistics provided by the probe can provide results, particularly in terms of CPU usage, that are not accurate to the actual amount of CPU utilization used by the monitored Virtual Machine. This is because CPU time is "stolen" by other instances and the top command can often return much lower than the external monitoring solutions. This problem can be solved by simply hosting Virtual Machines on dedicated hosts. Figure 4.7 uses an example of shared hardware to provide data for the most common usage of Virtual Machines on cloud hosting platforms. Over the monitored period the eight Virtual Machines returned consistently higher reported utilization on AWS Cloudwatch compared to the values returned by the probe. Despite this disparity, the difference in utilization is less than 1% for the low load system tested, as shown in Figure 4.7, meaning accuracy can be considered likely usable for most purposes.

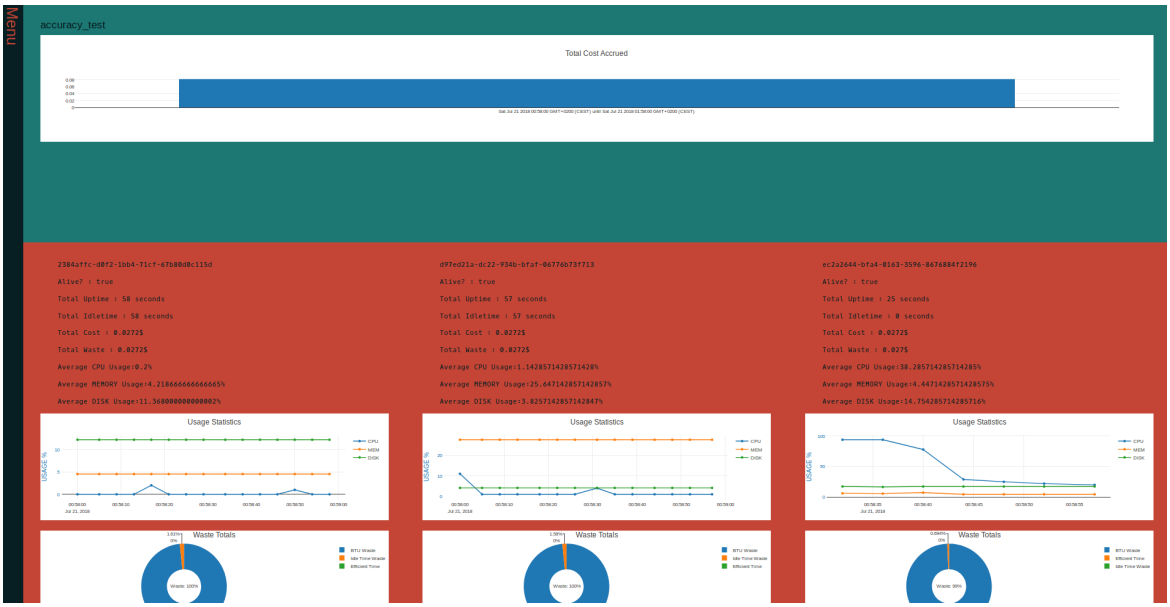

Fig. 4.8 A screenshot of the data from three simultaneously alive Virtual Machines of the same image type running on three different cloud Providers. From left to right: Azure, Google Cloud, AWS

4.3 Discussion

4.3.1 Requirements and Design Choices

Based on the service designed, the requirements are in-line with what should be expected of a cloud monitoring solution. This is because the requirements are based on literature that already exists on the subject. The requirements chosen are largely not limiting in interpretation which leaves room for future monitoring solutions to take slightly varied approaches to solving the same problem.

The requirements were rigid in some respects in the sense that they demanded, for example: a lightweight solution. Because of this there was almost no room for any other sensible choice in terms of probing than that that was explored by this project. Should for example some requirements be relaxed or altered there might be interesting implications in terms of design, for example a peer-to-peer solution not requiring a hosted service. This would obviously be something that the designer would have to consider whether they thought that was important enough to change.

4.3.2 Implementation

Ultimately, based on the evaluation of the implementation of the monitoring solution in this project, the results can be considered satisfactory when compared to the requirements. The solution does in fact monitor Virtual Machine instances without regard for their location or host and does so in a lightweight manner. In that way, the project is a success. The room for improvement come in terms of usability of the service designed. The low-horizontal-scalability of the service in its current state due to lacking optimization and the small breadth of utilization statistics monitored by the probe mean that as a tool the probe and service combination are not quite at the level of utility to potential users. The additions required are trivial in conception, but would require a not insignificant time investment to improve. As well additional features such as an API might make the service even more useful.

Because the requirements originally elicited for a monitoring service were reasonably decided upon, the design of the service took those requirements into account, and because the implementation of that design was functional and any shortcomings it experienced are clearly rectifiable, the monitoring solution created in this project fulfills the needs of the thesis.

Chapter 5

Conclusions

5.1 Summary

There exists a need for a cloud-monitoring solution that is more universal in scope, is easy-to-use, and also focuses on the cost and waste of cloud applications. Literature on cloud monitoring discusses different utilization statistics to be monitored in a cloud monitoring solution as well as some ways to calculate cost in cloud monitoring. Monitoring solutions exist already, but none meet all the requirements of being: non-proprietary, universal, and having a focus on cost and waste. A set of requirements based on existing literature and cloud monitoring solutions can be formulated. Those requirements can be interpreted as a design based on pushing data from probes to a service in a hub-and-spoke topology. This design can be implemented as a set of servers and databases hosted on public clouds and generated probes sending data to this service. The designed service-probe combination was effective, but left room for improvement in the form of future work.

5.2 Future Work

Although this project has successfully shown that a monitoring solution that meets the elicited requirements can and has been created, there is future work to ensure quality of the tool. This future work could include:

1. Expansion of the monitored utilization statistics to include for example: Network Utilization and the inclusion of this in the cost calculation.

2. The inclusion of an API in order to deliver data to users in a manually processable format.

3. The optimization of the data processing.
4. Reorganization of the user interface such that it is more user-friendly.
5. Account for variations in Billing Time Units for public cloud providers.
6. Testing for security vulnerabilities.
7. Reworking the system to allow other deployments of the service to be securely deployed without connection to the main service at https://universalcloudmonitoring.com.

References

[1] Giuseppe Aceto, Alessio Botta, Walter Donato, and Antonio Pescapè. Cloud monitoring: A survey. 57:2093–2115, 06 2013.

[2] Mohiuddin Ahmed, Abu Ali Ibn Sina, Raju Chowdhury, Mustaq Ahmed, and Md. Mahmudul Hasan Rafee. An advanced survey on cloud computing and state-ofthe-art research issues. 2012.

[3] Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Prem Prakash Jayaraman, Samee Ullah Khan, Adnene Guabtni, and Vasudha Bhatnagar. An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art. *Computing*, 97(4):357–377, Apr 2015. ISSN 1436-5057. doi: 10.1007/s00607-014-0398-5. URL https://doi.org/10.1007/s00607-014-0398-5.

[4] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to adapt applications for the cloud environment. *Computing*, 95(6):493–535, Jun 2013. ISSN 1436-5057. doi: 10.1007/s00607-012-0248-2. URL https://doi.org/10.1007/s00607-012-0248-2.

[5] ATIS. Atis telecom glossary: Network topology. URL http://www.atis.org/glossary/definition.aspx?id=3516.

[6] Ivona Brandic, Dejan Music, Philipp Leitner, and Schahram Dustdar. Vieslaf framework: Enabling adaptive and versatile sla-management. pages 60–73, 2009.

[7] Eddy Caron, Luis Rodero-Merino, Frédéric Desprez, and Adrian Muresan. Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds. Research Report RR-7857, INRIA, February 2012. URL https://hal.inria.fr/hal-00668713.

[8] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall. Toward an architecture for monitoring private clouds. *IEEE Communications Magazine*, 49(12):130–137, December 2011. ISSN 0163-6804. doi: 10.1109/MCOM.2011.6094017.

[9] Massimiliano Rak Dana Petcu, Ciprian Craciun. Towards a cross platform cloud api - components for cloud federation. *Conference: CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science*, 2011.

[10] B. Grobauer, T. Walloschek, and E. Stocker. Understanding cloud computing vulnerabilities. *IEEE Security Privacy*, 9(2):50–57, March 2011. ISSN 1540-7993. doi: 10.1109/MSP.2010.115.

[11] R. P. Karrer, I. Matyasovszki, A. Botta, and A. Pescape. Magnets - experiences from deploying a joint research-operational next-generation wireless access network testbed. In *2007 3rd International Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities*, pages 1–10, May 2007. doi: 10.1109/TRIDENTCOM.2007.4444714.

[12] Rouven Krebs, Christof Momm, and Samuel Kounev. Architectural concerns in multi-tenant saas applications. 04 2012.

[13] Jörn Kuhlenkamp and Markus Klems. Costradamus: A cost-tracing system for cloud-based software services. In *ICSOC*, 2017.

[14] P. Leitner, J. Cito, and E. Stöckli. Modelling and managing deployment costs of microservice-based cloud applications. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pages 165–174, Dec 2016.

[15] Asma Ben Letaifa, Amel Haji, Maha Jebalia, and Sami Tabbane. State of the art and research challenges of new services architecture technologies. pages 69–88, 2010.

[16] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

[17] J. Moses, R. Iyer, R. Illikkal, S. Srinivasan, and K. Aisopos. Shared resource monitoring and throughput optimization in cloud-computing datacenters. pages 1024–1033, May 2011. ISSN 1530-2075. doi: 10.1109/IPDPS.2011.98.

[18] Barbara Pernici, Pierluigi Plebani, and Monica Vitali. About monitoring in a service world. In Markus Helfert, Cornel Klein, Brian Donnellan, and Oleg Gusikhin, editors, *Smart Cities, Green Technologies, and Intelligent Transport Systems*, pages 3–23, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63712-9.

[19] RightScale. State of the cloud report. Technical report, 2018. URL https://www.rightscale.com/lp/state-of-the-cloud?campaign=7010g0000016JiA.

[20] B. Safaei, A. M. H. Monazzah, M. B. Bafroei, and A. Ejlali. Reliability side-effects in internet of things application layer protocols. In *2017 2nd International Conference on System Reliability and Safety (ICSRS)*, pages 207–212, Dec 2017. doi: 10.1109/ICSRS.2017.8272822.

[21] Nikhil Saswade, Vinayak Bharadi, and Yogesh Zanzane. Virtual machine monitoring in cloud computing. 79:135–142, 12 2016.

[22] J. Spring. Monitoring cloud computing by layer, part 1. *IEEE Security Privacy*, 9(2):66–68, March 2011. ISSN 1540-7993. doi: 10.1109/MSP.2011.33.

[23] Srikanth Sundaresan, Walter de Donato, Nick Feamster, Renata Teixeira, Sam Crawford, and Antonio Pescapè. Broadband internet performance: A view from the gateway. *SIGCOMM Comput. Commun. Rev.*, 41(4):134–145, August 2011. ISSN 0146-4833. doi: 10.1145/2043164.2018452. URL http://doi.acm.org/10.1145/2043164.2018452.

[24] VMWare. *Definition of Virtual Machine according to VMWare*. URL https://pubs.vmware.com/vsphere-50/index.jsp?topic=%2Fcom.vmware.vsphere.vm_admin.doc_50%2FGUID-CEFF6D89-8C19-4143-8C26-4B6D6734D2CB.html.

[25] Wikipedia, the free encyclopedia. Mesh topology diagram, 2018. URL https://en.wikipedia.org/wiki/Network_topology#/media/File:NetworkTopology-FullyConnected.png. [Online; accessed June 15, 2018].

[26] Wikipedia, the free encyclopedia. Star topology diagram, 2018. URL https://en.wikipedia.org/wiki/Network_topology#/media/File:StarNetwork.svg. [Online; accessed June 15, 2018].

[27] S. Zhang, S. Zhang, X. Chen, and X. Huo. Cloud computing research and development trend. In *2010 Second International Conference on Future Networks*, pages 93–97, Jan 2010. doi: 10.1109/ICFN.2010.58.

Code

Listing 1 An example of a generated bash probe

```
#!/bin/bash

##CONFIG##
USER_ID=5b1818145f2f3b51b3c5b0f4
IMAGE_TOKEN=5b461dea998e275f28faba88
DESTINATION="http://www.universalcloudmonitoring.com"
PORT=3000
PING_RATE=5
INSTANCE_TYPE=5b1ecf3c6bcc0a4d5d81aab7

##CODE - Do not modify##
function CPU_usage(){
        TOTAL_CPU_USAGE=0
        TOTAL_CPU=$(grep -c ^processor /proc/cpuinfo)
    #set number of CPUs to check for
        declare -a 'range=({"0..$TOTAL_CPU"})'
        let "TOTAL_CPU=$TOTAL_CPU - 1"

        #declare array of size TOTAL_CPU to store values
    #(eg. 8 cpus makes arrays of size 8)
        declare -a PREV_TOTAL=( $(for i in ${range[@]};
    do echo 0; done) )
        declare -a PREV_IDLE=( $(for i in ${range[@]};
    do echo 0; done) )

        for i in {1..3}; do
            SUM=0
        declare -a 'range=({"0..$TOTAL_CPU"})'
        for j in ${range[@]};do
            CPU=(`cat /proc/stat | grep "^cpu$j "`)
            # Get the total CPU statistics.
            unset CPU[0]
            # Discard the "cpu" prefix.
            IDLE=${CPU[4]}
            # Get the idle CPU time.
```

```
            # Calculate the total CPU time
            TOTAL=0
            for VALUE in "${CPU[@]}"; do
                let "TOTAL=$TOTAL+$VALUE"
            done

            # Calculate the CPU usage since we last checked.
            let "DIFF_IDLE=$IDLE-${PREV_IDLE[$j]:-0}"
            let "DIFF_TOTAL=$TOTAL-${PREV_TOTAL[$j]:-0}"
            let "DIFF_USAGE=(1000*($DIFF_TOTAL-$DIFF_IDLE)/
            ($DIFF_TOTAL+5))/10"
            let "SUM=$SUM+$DIFF_USAGE"
            # Remember the total and idle CPU times
            for the next check.
            PREV_TOTAL[$j]="$TOTAL"
            PREV_IDLE[$j]="$IDLE"
        done
            let "SUM=$SUM/($TOTAL_CPU+1)"
            let "TOTAL_CPU_USAGE=$TOTAL_CPU_USAGE+$SUM"
            sleep 1
        done
        let "TOTAL_CPU_USAGE=$TOTAL_CPU_USAGE/3"
    echo $TOTAL_CPU_USAGE
        return $TOTAL_CPU_USAGE
}

function send_data (){
        curl --fail --header "Content-Type: application/json" \
                --header 'Expect:' \
                --request POST \
                --data "$1" \
                "$DESTINATION/probePost"
        res=$?
        return $res
}

#sends json data via curl to the probePost
while [ true ]; do
        sleep 1
```

```
    filename="tempStorage.json"
    fail=0
    lineNumber=1
    while read -r line
    do
        json="$line"
        send_data "$json"

        res=$?

            if test "$res" != "0"; then
               break
            else
                #delete lines of data already transfered
                sed -i "$lineNumber"'s/.*//' "$filename"
            fi
            let "lineNumber++"
    done < "$filename"
    #Clear the deleted lines at the end
    sed -i '/^\s*$/d' "$filename"

    #Gather new data
    memFree=$(awk '/MemFree/ {printf( "%f\n", $2)}'
/proc/meminfo)
memTotal=$(awk '/MemTotal/ {printf( "%f\n", $2 )}'
/proc/meminfo)
MEMORY=$(free -m | awk 'NR==2{printf "%.2f",$3*100/$2 }')
#Shows memory usage without buff/cache included
diskSize=$(df --output=size -B 1 "$PWD" |tail -n 1)
diskUsed=$(df --output=used -B 1 "$PWD" |tail -n 1)
DISK=$(awk "BEGIN {printf \"%.2
    \",${diskUsed}/${diskSize}*100}")
CPU_usage
CPU=$?
UUID=$(dmidecode | grep -i uuid | awk '{print $2}' |
tr '[:upper:]' '[:lower:]')
TIME=$(date +%s)

    #This is where new data will be extracted and sent
```

```
        newData='{"oauthid": "'$USER_ID'",
                  "app":"'$APP_TOKEN'",
                  "image":"'$IMAGE_TOKEN'",
                  "uuid":"'$UUID'",
                  "cpu":"'$CPU'",
                  "mem":"'$MEMORY'",
                  "disk":"'$DISK'",
                  "time":"'$TIME'",
                  "instance_type":"'$INSTANCE_TYPE'"
        }'

        send_data "$newData"
        res=$?
        if test "$res" != "0"; then
           echo $newData >> "tempStorage.json"
        fi

        :
done
```

Data for comparing monitoring data from the probing solution and Amazon Cloudwatch. This data was collected over an hour of probing and is represented in chart form in Figure 4.7.

CPU Monitoring (w/ probe)	CPU Cloudwatch (w/ Probe)	CPU Cloudwatch No Probe	Instance Number
0.63%	1.06%	0.88%	Instance 0
0.64%	1.02%	0.88%	Instance 1
0.66%	1.04%	1.00%	Instance 2
0.70%	1.03%	0.99%	Instance 3
0.70%	1.00%	0.99%	Instance 4
0.71%	1.18%	0.99%	Instance 5
0.75%	1.07%	1.00%	Instance 6
0.81%	1.03%	0.83%	Instance 7

Data for comparing monitoring data from Amazon Cloudwatch when the probe is running normally compared to when the probe cannot send its data. This data was collected over an hour of probing and is represented in chart form in Figure 4.6.

CPU Monitoring (w/ probe)	CPU Cloudwatch (w/ Probe failed)	Maximum Utilization	Instance Number
1.06%	1.30%	2.83%	Instance 0
1.02%	1.24%	3.05%	Instance 1
1.04%	1.31%	2.99%	Instance 2
1.03%	1.29%	2.99%	Instance 3
1.00%	1.31%	2.99%	Instance 4
1.18%	1.29%	2.99%	Instance 5
1.07%	1.26%	2.83%	Instance 6
1.03%	1.31%	2.95%	Instance 7