Monitoring Latency Sensitive Enterprise Applications on the Cloud

Shankar Narayanan
Purdue University
spuzhava@purdue.edu

Ashiwan Sivakumar
Purdue University
asivakum@purdue.edu

ABSTRACT
Enterprises are increasingly deploying their applications in the cloud given the cost-saving advantages and the potential to geo-distribute applications to ensure resilience and better service experience. Most of the enterprise applications have stringent latency constraints for better user experience or by business requirements. One of the fundamental challenges in the deployment of such geo-distributed enterprise applications is to monitor their performance once deployed. On the one hand, it is clear that user observed latency is the most important parameter and enterprises should be able to efficiently measure the end-to-end latency at the granularity of every user transaction, on the other hand, they should also be able to isolate the problem to each component to optimize cost and improve utilization. It is therefore crucial for any monitoring framework to provide latency measurements at both the levels of granularity. Also the monitoring framework should be easy to integrate with any application and have minimal impact on the application in order to be scalable. In this work, we present the design and implementation of such a monitoring framework for latency sensitive applications on the cloud. We also evaluate our framework using benchmark applications deployed on two prominent cloud service providers, Amazon AWS and Microsoft Azure.

1 Introduction
Cloud computing promises to reduce the cost of enterprises as they can optimally lease the required amount of compute and storage resources from cloud providers. It allows them to scale their resources based on the application load and usage patterns. This helps them substantially lower their capital and operational expenses. Cloud computing also allows the enterprises to deploy their application closer to the users thereby improving the user experience in terms of access latency. The flexibility offered by cloud providers to dynamically determine the scale and location of the application, makes it possible for enterprises to scale and deploy multiple instances of the application independent of each other as suggested by Hajjat et al [18]. Moreover, enterprise applications require a high degree of availability and scalability making it naturally suitable for cloud environments.

Enterprise applications are typically multi-tiered and service-oriented that have complex business requirements and often involve interaction among multiple components for processing a given user transaction. Figure 1 shows the architecture of StockTrader which is a benchmark enterprise application used to evaluate the performance and inter-operability of service oriented architecture (SOA). Like all enterprise applications following the established MVC architecture, StockTrader has a front-end, a business logic, a configuration service, an order processing service and a back-end component. Figure 2 shows a sample deployment of the application highlighting the complex interactions between the various components of the application.

Cloud environments are susceptible to performance problems that can last anywhere between a few minutes and a few days. Also, these problems can occur
Figure 2: StockTrader Deployment Graph

Figure 3: A CDF showing the database component latency in the US North data-center of Windows Azure over a duration of four hours for two days. Day1 shows poor response time values over Day2.

at a specific fault domain or could span over an entire data-center. For instance, we observed a performance issue with the entire data-center that lasted for nine days and was reported to affect a few domains randomly within the data-center. In Figure 3 we present a CDF showing the latency of requests served by a database component deployed in the US North data-center of Microsoft Azure. This data was collected for a duration of about four hours across two days. We can see that the performance of the component varies drastically across days. These observations emphasize the fact that cloud environment is highly dynamic and prone to failures. These issues have severe impact on the user performance and therefore applications deployed on the cloud need to be monitored continuously. While it is important, it is not merely sufficient to monitor the end-to-end user performance, since the problem could be specific to a given data-center and possibly affect only a subset of components. Abandoning an entire deployment in such cases would lead to poor utilization of cloud resources or may lead to increased cost.

Tools like myARM, HTTPerf that measure end-to-end latency at the front-end server are ineffective for the reasons mentioned above. Most cloud platforms provide generic performance counters which monitor VM performance like CPU usage and memory usage, but such metrics can at best be used as performance indicators. For instance it is not possible to predict a transaction’s performance based on the CPU or memory usage observed at the components. Code profilers and OS provided tracers like Dtrace, JProfiler are not flexible and do not provide measurement at user-level and component-level granularities. Tracing frameworks like X-Trace, Magpie and ETW [17, 13, 12] provide far too detailed information to make them scalable. Moreover, we have observed that these solutions involve substantial development effort to be integrated with the application. Therefore, it is evident that the monitoring framework needs to be fast, scalable and easy to integrate with any enterprise application.

In this work, we present the key insights for the design of such a monitoring framework and show the benefits of our design on a benchmark enterprise application deployed on two major cloud providers namely, Amazon AWS and Microsoft Azure. Existing solutions like Magpie, X-Trace or ETW can provide transaction level performance, but are not scalable and require substantial development effort to integrate and customize for the above requirements. The design principles of our framework is similar in spirit to that of X-Trace, but abstracts out information that is not relevant to our purpose. This enables aggregation of measurements, making our design much more scalable. Also, we come up with a formal specification of measurements for such applications that helps automating the instrumentation process and requires minimal development effort. Since we designed our framework based on extensive experiments on real cloud environments, our implementation presents a simple and scalable framework to measure the performance of geo-distributed applications in the cloud.

2 Design Goals

The distributed deployment of enterprise applications and the complex interaction between their components present several key challenges to the design of a monitoring infrastructure for such applications. First, the monitoring needs to be performed at multiple levels of granularity. While the user observed latency is critical for measuring end-to-end performance, component level latency is essential for problem isolation and cost effectiveness. The design of the monitoring system should therefore be able to provide both these abstractions.

Second, the monitoring framework itself should be scalable. As mentioned earlier, enterprise applications are deployed in a cloud environment primarily to en-
joy the benefits of elasticity and scalability of the application and its components. Hence, the monitoring framework should also be able to adapt to such dynamic changes in the application deployment. Also, the monitoring framework should not impose any load on the application or impede the performance of the application.

Finally the monitoring tool should be easy to integrate with the application and should require minimal development effort. It should not require significant changes to the application and yet be able to obtain the relevant metrics that are essential for analyzing the end-to-end and component level performance of the application. The infrastructure should provide a generic interface that can be adapted easily to any standard enterprise application.

3 Design

In this section we present the key insights that lead to the design of our monitoring infrastructure. We begin by describing the overall architecture of the framework and then present our measurement model that forms the core of our design. We then discuss in detail the experiments and observations that guide the design of the various components of the framework.

3.1 Framework Architecture

Figure 4 shows the overall architecture of the monitoring framework. There are three major components that constitute the framework: the instrumented application, log server and global collector. The architecture is similar in spirit to that of the X-Trace except that each component has a different implementation optimized based on the design insights that are described in §3.2.

3.2 Measurement Model

As described in §2, the key requirements for any monitoring framework is speed and scalability. Unfortunately, most of the existing tracing and monitoring tools like X-Trace or ETW track too much detail for them to be scalable. Intuitively, it should be possible to create a faster and more scalable solution if some of these details can be abstracted away. In this section, we construct a minimalistic model that defines the set of parameters that needs to be tracked for computing the component and the end-to-end latency. As we shall see in §4, our model allows an implementation that abstracts information in the form of local and global aggregation. This construction reduces the load on the Collection Framework and makes the system more scalable.

To construct the model, we first need to identify all the communicating end points for every component $C_i$ in the application. Figure 5 shows a sample of our model with three components for readability, but it should be noted that this can be extended to any application with any number of components.

**Track time-period (not individual timestamps)**: As mentioned in §2, our aim is to measure the latency of every component, the link latency between the components and also the end-to-end latency for every user transaction. However most of the existing tools track events at every end-point which is clearly an overkill for our purpose. Since we are only interested in latency and not absolute time, it is sufficient to measure the time period $T_{i,j}^k$, where $i, j$ are the components that communicate and $k$ represents the $k^{th}$ call between the components $i, j$. This construction guarantees a performance benefit of 50% by reducing the number of events that need to be logged by half.

**Decouple the components**: The key factors that affect the scalability of any system are dependence and communication among its entities. It can be seen from our model that a component $C_i$ need not communicate its measurements to either its upstream or downstream
component. The Collection Framework (collector) can aggregate it based on the required granularity. The only glue parameter that needs to be communicated by the instrumentation framework is a global UID for each user transaction that aids the Collection Framework in grouping the measurements and forming an end-to-end summary. It should be noted that our model like X-Trace, provides flexibility for grouping and aggregation to happen at any level of the transaction. The key difference however is that X-Trace requires the entire set of measured values from all the components at a centralized location before it can group or aggregate the results. Our model on the other hand allows aggregation to happen at all levels: at the application, log server or at the global collector.

For instance, any component \( C_i \) that communicates to a downstream component \( C_{i+1} \) can aggregate the latency \( T^k_{i,i+1} \) measured across the \( k \) calls it makes to the \( C_{i+1} \) within the period \( T^k_{i-1,i(i)} \) (for a given \( k \) in component \( C_{i-1} \)). Also every component \( C_i \) would measure its end-to-end latency \( T^k_{i-1,i(i)} \) (for a given \( k \) in component \( C_{i-1} \)). This value would be used by the collection framework to measure the link latency between \( C_{i-1} \) and \( C_i \).

**Aggregate anywhere anytime:** One of the key advantages of our model is that it captures the latency as a time-period that can be aggregated at any point. For instance, consider the problem of determining the link latency between components \( C_i \) and \( C_{i+1} \). From figure 5, it can be seen that these latencies would be measured individually at \( C_{i+1} \) as \( (T_{i,i+1(i+1)}) \) and as an aggregated value \( (T^1_{i,i+1(i)} + T^2_{i,i+1(i)}) \) at \( C_i \) (since it is a downstream measurement). The collection framework would then compute the link latency between \( C_i \) and \( C_{i+1} \) by using the aggregate measured at \( C_i \) and subtracting the summed up value of the individual measurements \( T_{i,i+1(i+1)} \) made at \( C_{i+1} \). This clearly shows the simplicity of our model and its flexibility that allows even partial aggregation at all levels. For instance, the log server could aggregate the component latency and ship them to the storage at any point of time. It is important to note that correctness is maintained even if the measurements are partially written to the storage since addition is associative and the summary is generated globally after the user transaction is guaranteed to have completed. As a final observation, it should be noted that downstream components benefit lesser from aggregation (For example \( C_{i+2} \)) since they have no notion of aggregation and all measurements would have to be communicated to the collector.

Based on the above discussion we come up with the following equations for computing the Component Latency (CL) at each component \( C_i \) and the Link Latency (LL) computed globally:

\[
CL_i = T^1_{i-1,i(i)} - \sum_{j=1}^{N} \sum_{k=1}^{n} T^k_{i,j(i)} \text{ where } N \text{ is the number of components } C_i \text{ communicates with and } n \text{ is the number of calls } C_i \text{ makes to each of the other components (the value of } n \text{ can vary for each } j).
\]

\[
LL_{i,i+1} = \sum_{k=1}^{n} T^k_{i,i+1(i)} - \sum_{k=1}^{n} T^k_{i,i+1(i+1)} \text{ where the first term is obtained from the aggregate log written by } C_i \text{ and second term is computed at the collector by aggregating the individual entries written by } C_{i+1}.
\]

It is clear from the above equations that our model makes the measurements at each component independent of the other components. This abstraction also helps aggregation of measurements at every level in the monitoring framework which makes the solution more responsive and scalable.

### 3.3 Design of the monitoring framework

In this section, we present our design of the various entities that constitute the monitoring framework as seen in §3.1. The design of our monitoring framework is primarily guided by the model described in §3.2 and based on extensive experiments that we carried out on two major cloud providers - Amazon AWS and Microsoft Azure [1, 9].

**Instrumenting the application:** The framework consists of an instrumented application that generates logs at the end points of the components. Ability to integrate the instrumentation entity of the framework with the application is a non-trivial task and development intensive. For instance, instrumentation using the X-Trace involves changes similar to that of a custom hand-coded solution as mentioned in §5. Reducing this development effort is one of the key design goals of our framework, which has been achieved by making use of aspects.

Aspects are used to modularize concerns like Logging that cross-cut multiple objects. The functionality provided by the aspect is separated from the application code and can be compiled together at a later point of time. This helps in generating reusable code which can be easily integrated with any application. By using aspects, we have developed reusable instrumentation code, whose generation can be automated if the developer provides the name of the methods of each application components.

**Aggregation using Log Server:** Our measurement model not only simplifies the instrumentation by abstracting out irrelevant information, but also helps distill information that can benefit from aggregation at all levels in the monitoring framework (§3.2). As shown in figure 4, we take advantage of this feature provided by our model to implement aggregation closer to the ap-
application component by using a log server that is deployed locally. The primary function of the log server is to off-load the burden of aggregation from the application. The log server maintains an aggregation map and periodically writes them to the permanent storage which is a part of the collection framework.

**Global Collection Framework:** The *Collection Framework* is a global entity that does the final level of aggregation of the measurement logs and forms the summary report of the component and link latencies per user request. The collector also computes the end-to-end user observed latency which is one of the most important requirements of any monitoring framework that is used to monitor enterprise applications deployed on the cloud.

The *Collection Framework* comprises of the notification queue, the storage and the global collector that are shown in figure 4. As mentioned above the *Collection Framework* is responsible for the final aggregation and generating the summary report for every user transaction. The collector is nothing but a process that collects the measurements distributed across various storage entities (usually deployed locally to the component to minimize write time), performs aggregation and generates transaction summary using these measurements. Since there is no dependency across multiple user transactions, the collector can be deployed in one or more VM’s on the cloud based on the load and response time constraints as long as each collector process takes complete ownership of a transaction.

The notification queue serves three important purposes. First, it acts as a notification to the collector that the user transaction has been completed and is ready to be processed at the collector. This notification can be generated by any deterministic event happening across the application. One such event could be when the user response is shipped back to the user. The log server receives this deterministic event and pushes the notification into the queue. Second, using a queue implementation helps decouple the log server from the collector. This allows the log server to scale with the application component by using a log server that is deployed locally. The primary function of the log server is to off-load the burden of aggregation from the application. The log server maintains an aggregation map and periodically writes them to the permanent storage which is a part of the collection framework.

**Figure 6:** Instrumentation of the application

log server minimizes the number of writes quite drastically as corroborated by the results shown in §5.

### 4 Implementation

In this section, we discuss the design goals in detail and the implementation of different parts of the measurement framework. The measurement framework can be logically sub-divided into the instrumentation, logging and collection entities. Under each sub section, we first discuss the rationale behind the solution and then the implementation details.

#### 4.1 Instrumentation framework

In order to be simple and scalable, any instrumentation framework must closely follow the model shown in Figure 6. First, we should identify all the function end-points that are to be instrumented to get the computation time and communication latency (Specification for the application end-points). Second, the application should decide on the measurement metric specifications which are to be reported by the instrumentation framework. Finally the framework should get the log format specification understood by the logging framework and generate logs based on all the three inputs. With this model in mind, we present different solutions for the instrumentation framework and the motivation behind choosing the solution that we have implemented.

We present a suitability matrix in figure 7 comparing the existing tools based on our requirements. The various criteria for comparison which we consider include easy portability, flexibility, development effort and generic nature. From the matrix we can observe that existing tools like X-Trace, Dapper provide transaction level performance measurements, but are developed for customized environments and platforms. They are more flexible as they provide far too detailed information which can be tailored to any level, but this makes them difficult to scale on cloud environment. The performance monitoring counters provided by the application servers like the IIS counters are supported for smaller subsets of .NET language framework. They provide information at course granularity
and require substantial effort to interpret the data. But they are generic and application agnostic. On the other extreme, a full-fledged custom-coded instrumentation framework provides information at any level as they are under the developer’s control. This framework can scale along with the application instances as they are developed by refactoring the application code. But this solution poses significant development effort and is not easily portable to different applications.

We evaluated state-of-the-art distributed call tracing tools like the X-Trace, Dapper and ETW [17, 20, 12]. X-Trace is a highly resilient and distributed call tracing tool. It logs information on an asynchronous event based model. X-Trace requires a metadata to be sent along with the request, for which every function call has to be modified. It sends collected measurements to a centralized reporting server which is not scalable. Though there are other scalable collection mechanisms available for X-Trace, like the Flume and HBase for HDFS, they require integration of many tools. A simple solution would be to save the logs in a file and send it to our own collection framework. But using X-Trace directly to do this would require similar coding effort as with a handcrafted solution. One main disadvantage of X-Trace is that the application and any underlying protocol has to be modified to carry X-Trace metadata along the data path. Dapper provides measurement data at a finer level of granularity, since it has been implemented at the RPC layer in Google’s cluster. But it is tailored for the local RPC unit and it does not explicitly capture sequential and parallel activity.

Since our motive is to use a measurement framework that would require minimal coding effort and minimal application interference, we initially chose to write interceptors and filters. Filters and interceptors use Aspect Oriented Programming [21] to intercept function calls. Filters are written for web containers (front ends) and Interceptors are available for EJB containers (Business logic components). Eventhough they provide separation of instrumentation concern from the application, they can be used only with standard java technologies like Servlets and Enterprise Java Beans. The information which we get from interceptors and filters based on our measurement model are just the end-to-end time taken at the front-end and the business-logic components. It is difficult to separate inter-component communication latency from the component computation time. One solution would be to write our own aspects to log inter-component communication endpoints together with filters and interceptors. But this would involve substantial development effort to use some binding mechanism to interpret and associate the logs collected from the filter, interceptor and the custom log components with the user session. This would also require the generation of the binding information like the UID for every user session, source and destination components, name of the data-centers etc., from the application.

From the above discussion it is evident that aspects can be used to modularize concerns that cross-cut multiple objects. In our case, the logging functionality can be written as aspects, independent of the application and can be later compiled together with the application source code. We also understand the necessity for a binding mechanism to associate the data with user session. X-Trace is a robust tool and it already provides us with the binding mechanism (X-Trace TaskID) which can be used to bind logs from various application components with the user session. If X-Trace logging code can be used in conjunction with aspects to design a highly modular framework, then it would make the instrumentation framework easy to integrate with any application. But X-Trace provides highly extensive tracing details which are not required for monitoring enterprise applications in the granularity of user requests. X-Trace logs every instrumentation point as event to the central server. According to our measurement model, it would be highly beneficial to aggregate the logs at every component level.

As shown in Figure 6 our instrumentation framework gets the logging end-points from the application and generates an aspect code using X-Trace APIs. The X-Trace metadata has been customized based on the measurement metric specification provided by the application. There is a mapper which translates the X-Trace report message into a log message whose specification is provided by the logging framework. The aspect code is modular and can be easily integrated with any application. Our instrumentation framework is highly resilient and convenient where as X-Trace used directly, provides only resilience.
4.2 Logging framework

The Logging framework has to collect the logs from the application component and aggregate before handing over the logs to the collection framework which will write to the local storage. This functionality can be provided by a server process which will receive the logs from the application and aggregate the logs based on the measurement model. The logging server has to be scalable and must not be a bottleneck on the application. X-Trace tool provides a proxy server which gets the report messages from the application and sends it to a central server. This proxy server helps X-Trace instrumented application to send log messages asynchronously from the data path. Also the proxy server helps in sharing the load on the central server and prevents single point of failure to certain extent. The idea behind our logging framework is similar in spirit to that of X-Trace. But it provides additional advantage of aggregating the logs at every component so that the number of writes made to the local storage in the cloud can be minimized. As discussed in the design section, storage writes are costly and the latency is an additional overhead to the monitoring framework. Our logging framework benefits the whole monitoring process by aggregating logs at component-level and writing summarized logs to the storage.

It maintains a hash map of key-value pair indexed by the request UID or X-Trace TaskID in our case. Everytime the server gets a log message, it aggregates the values retrieved from the hash map with the new values and writes it back. This way even if the log message gets delayed from the application, the server would be capable of aggregating it. The server writes the logs to the storage once every 5 seconds. So even if the log messages take beyond 5 seconds to reach the proxy server, the collection framework will be able to aggregate the values based on the UID at a later point of time. Since memory is a bound resource, we clear old hash map entries at the proxy server every 30 seconds. Both the write time and resource free time are configurable parameters and they would not affect the correctness of the measurement data due to the binding mechanism and the associative nature of sum operation.

4.3 Collection framework

The collection framework should be scalable and fast with minimal work to be done by the application. Therefore the application should write the measurement logs to the local data-center. For policy reasons, some components of the application have to be in different data-centers. For example, storage or DBs might be common across applications deployed in multiple data-centers. Also some components of the application might be shared with other applications, which might be deployed in a different data-center. Hence we would require consolidation of measurements from multiple data-centers. Collection framework needs to know all the data-centers in which the application and its components are deployed. However, it is important to note that it is not necessary for all requests to use all the components in the application. The collection framework needs to track when a request-response chain is completed at the application. This can be deterministically done only when the application hands off the response back to the user at the front-end component.

There are different solutions possible for the collection framework using the different storage services offered by cloud providers. First, the application components can write to queues in the local data-center and the consolidation framework can pick from the queues. With this model, it is difficult to track when a request-response chain is completed. This would require the individual processes of the consolidation framework to interact among themselves to collate the information. Second solution would be to directly send the message to a collating process through network sockets. But this would hinder the decoupling of the producer and the consumer. Also we won’t have a presistent storage for the log messages. Third solution would be that the application component writes the logs to the local storage and then front-end indicates the completion of a request-response chain by enqueuing a message in the local queue which can be dequeued by the collection process. After extensive experiments on real cloud deployments, we implemented the third solution since it proves to be scalable by decoupling the components and provides a persistent storage.

In general, our implementation makes use of the queue service of the cloud providers. Microsoft provides the Windows Azure Queue and Amazon provides the SQS service. The use of queues helps the framework scale independent of the application since all the servers involved in the framework use the same end point of the queue. The queue, by itself is a reliable and scalable service provided by all major vendors [11, 4]. Also queues help in decoupling the producer and the consumer components. We leverage this feature to enable the framework to adapt and scale in tandem with the application and load. The front-end writes to the local queue once a request-response chain is completed. We also leverage the storage service provided by the cloud providers for the application components to write the logs. Microsoft provides the Azure Blob and Amazon provides the S3 service [10, 3]. We have implemented the collection framework on both Microsoft Azure and Amazon AWS platforms.
5 Evaluation

In this section, we evaluate the importance and effectiveness of our monitoring framework. We integrated our instrumentation framework with the DayTrader [5] and deployed the same on Amazon AWS along with the entire monitoring framework. DayTrader is a SOA application from the Apache stonehenge project [6] and has an architecture similar to that of StockTrader. We present our evaluation beginning with the analysis of cloud services used by the collection framework. We then show the benefits of accumulation and aggregation over existing tools. Finally, we compare the effort required for integrating our instrumentation framework with that of a custom hand-crafted solution.

5.1 Analysing performance of cloud storage

In §3, we presented our design of the collection framework. It can be seen from Figure 4 that our design makes use of two important infrastructural services provided by the cloud service providers, namely the storage and the queue services. It is evident that the performance of our monitoring framework is constrained by the performance of these services. In this section, we present our insights and observations on the performance of these services based on extensive experiments performed on two major cloud service providers namely Microsoft Azure and Amazon AWS. Our experiments in principle follow the architecture of our framework. However, our purpose is to exhaustively evaluate all design possibilities and therefore we deployed our experimental set-up on all the 12 data-center regions provided by AWS and Azure. These regions are distributed two apiece in the US, Europe and Asia by both the cloud providers. It should be noted that we performed all our experiments approximately for a period of 26 hours on both Azure and AWS. The experiments therefore reflect the performance results of about 100,000 writes to the storage and 17,000 messages to the queues.

5.1.1 Deployment set-up

The set-up for our evaluation of the performance of storage and the queue services is representative of an actual deployment of our framework involving three major components:

1. Generator
2. Storage, Queues
3. Collector

The Generator is a process representing the log server co-located with the application component that writes the aggregated measurement logs to the storage. However, as mentioned above our purpose is to exhaustively evaluate all design possibilities and therefore the generator process was deployed on all the 12 data-center regions and the measurement logs were written to the storage and queue in all the data-centers. Similarly the queue notifications were also sent by the generator process to queues in all the data-centers.

The storage and queue are services provided by the cloud providers which form a part of the cloud infrastructure. The Storage is a persistent storage service that is called blob and S3 in Azure and AWS respectively. The Queue is a scalable implementation of a FIFO service that is called AzureQueue and SQS in Azure and AWS respectively.

Finally the Collector is a process that corresponds to the Global collector in the framework architecture that takes out the notification from the queue and reads the measurement logs from the storage and aggregates them to form the summary. Similar to the Generator, the Collector was deployed in all the data-centers and reads the measurement logs from all the data-centers.

A key advantage of performing all possible combinations of reads and writes from all the data-centers is that we would be able to identify any asymmetry in the performance of these components across the data-centers. Also, this enables us to compare the performance of each activity across the two cloud providers.

5.1.2 Performance of Storage and Queues

Figure 8 shows the CDF of the latency of writes, reads and delete activities to both the blobs and queues in US-North data-center from Generator and Collector processes deployed locally within the same data-center. The graph shows that writes to the queue are the most expensive operation followed by writes to the blob storage.

Figure 9 shows the CDF of the latency of writes, reads and delete activities to both the S3 and SQS in US-West data-center from Generator and Collector processes deployed locally within the same data-center. Here, we observe that the reads from the Queues are the most expensive operation followed by the writes to the S3 storage. From these observations we can infer
that our design should try to minimize the queue operations for better performance.

Figures 10 and 11 show the CDF of the latency of reads and writes to the Azure Queues located at all the six data-center regions from a Collector and Generator located at the US-North data-center. As expected, the local reads (USN) are the least expensive while reads from Queue in Asia are the most expensive. This confirms our intuition that queues are too expensive an entity to support a design that uses only queues to carry the logs.

5.2 Impact of instrumentation framework on the application

We evaluate the impact of our instrumentation on the application and compare it with a custom hand-coded solution. In §4, we presented a comprehensive analysis of the possible solutions based on our design goals like easy integration, flexible measurement and generic nature. The framework we implemented uses X-Trace to instrument the application, but the functionality is separated from the application by the use of aspects. For inter-component function calls which use protocols like SOAP, the signature of the methods have to be changed inorder to carry the X-Trace metadata as an argument. Otherwise, the instrumentation has been made generic through the extensive use of aspects. An intuitive measure of the impact of an external framework is the effort required in integrating the tool with the application. We measure the development effort required to integrate the instrumentation with the application in terms of the Lines Of Code (LOC) which is the standard measure of effort in software engineering.

<table>
<thead>
<tr>
<th>Category</th>
<th>Handcrafted (A)</th>
<th>X-Trace with Aspects (B)</th>
<th>%benefit (B over A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same</td>
<td>15250</td>
<td>15250</td>
<td>0%</td>
</tr>
<tr>
<td>Modified</td>
<td>593</td>
<td>465</td>
<td>20%</td>
</tr>
<tr>
<td>Added</td>
<td>878</td>
<td>166</td>
<td>80%</td>
</tr>
<tr>
<td>Automated</td>
<td>0</td>
<td>166</td>
<td>27%</td>
</tr>
</tbody>
</table>

Figure 12: A table showing a comparison of the new lines of code added and SLOC for the instrumentation tool with that of other solutions.

Figure 12 shows the table comparing the LOC for our instrumentation implementation with that of a custom hand coded solution. The table shows that around 27% of the code added for instrumenting the application can be automated. But this just includes the code which is separated from the application using aspects. Since aspect oriented programming does not provide a solution to change the signature of method calls during compile time, we have to explicitly modify all the function calls in the control flow of a request from the front-end to the business service and from the business service to the database backend, to include the X-Trace metadata object as an extra argument. This also involves changing the xml specification files like the WSDL document for SOAP protocol. But the actual code to instrument the application is made generic using aspects across all the application component. We can also see that the new lines of code added is reduced by around 80% when compared to a custom coded solution. The most important point to be noted here is,
this code is completely reusable with any application. From the above discussion, we can observe that modifying method signature to carry the X-Trace object as an argument is the only constraint to a completely automated solution. This problem exists with any instrumentation solution for enterprise applications due to the binding data which should be passed inband with the request. One solution to avoid this problem is to introduce new remote methods using java RMI and every time before a trade operation is performed, the client (business logic component) can request for the X-Trace metadata from the server (front-end). This code can be implemented using aspects and can be reused for any application. But this implementation is beyond the scope of this project.

5.3 Benefits of accumulation

In this section we evaluate the benefits of the accumulation that is performed by the log server. As discussed in §3, the log server naturally performs accumulation of the measurement logs during the process of aggregation. However, the benefits we derive due to the accumulation are quite substantial when compared to the native X-Trace which logs every event to the centralized server. This results in many writes happening to the permanent storage which is clearly an overhead given our design goals. As seen from the table in the Figure 13, the benefits that we derive from the process of accumulation is significant given that writes to storage is an expensive operation in the cloud. The table shows the writes made to the storage for different types of trade operations performed by a random user from the DaCapo benchmark [16]. It shows that in this case we save around 78% of the writes made to the storage. We have to note that more the number of transactions, more is the benefit observed by aggregation.

<table>
<thead>
<tr>
<th>User request type</th>
<th>Storage writes without aggregation</th>
<th>Storage writes with aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FE</td>
<td>BS</td>
</tr>
<tr>
<td>Login</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Portfolio</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Update profile</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Home</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Buy</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Sell</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Account</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>24</td>
<td>40</td>
</tr>
</tbody>
</table>

Figure 13: DayTrader accumulation benefits for a random user from the DaCapo benchmark for all types of trade operations performed.

5.4 Impact of aggregation on the cost of data transfer

In this section we evaluate the benefits of aggregation with respect to data IO and prices charged by Microsoft Azure and Amazon AWS for data storage and transfer. The benefits we derive due to the aggregation are quite substantial when compared to the native X-Trace used directly. We present our analysis on the benefits of aggregation in Figure 14. The table shows a comparison between native X-Trace and our logging framework with aggregation. The analysis is done for the same experiment described in §5.3, involving the transactions made by a random user from the DaCapo benchmark. There are few assumptions made:

1. The 100,000 sessions initiated by the user involve all the 7 types of trade operations.
2. The size of all log messages is 1KB, as observed in the logs generated by DayTrader.
3. The above per day trend continues for a month.

It is seen that with aggregation, we obtain around 75% reduction in the data written to the storage. We have also tried to analyze the reduction in the data storage and data transfer costs due to aggregation based on the pricing schemes of Microsoft Azure and Amazon AWS [8, 2]. From the table it is evident that for the above random user, aggregation of log messages help reduce the data storage and data transfer costs by around 76% compared to native X-Trace.

<table>
<thead>
<tr>
<th></th>
<th>Native X-Trace</th>
<th>Proposed framework with aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes written (MB)</td>
<td>6400</td>
<td>1600</td>
</tr>
<tr>
<td>(100,000 messages per day)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage saved</td>
<td>N/A</td>
<td>75</td>
</tr>
<tr>
<td>Approximate cost ($) (per month)</td>
<td>50</td>
<td>12</td>
</tr>
<tr>
<td>Percentage saved</td>
<td>N/A</td>
<td>76</td>
</tr>
</tbody>
</table>

Figure 14: A table showing the benefits of aggregation with respect to percentage reduction in data IO and data transfer cost saved as opposed to native X-Trace

6 Related Work

Prior works have pointed out the presence of performance problems with the cloud [19, 15, 14]. In this paper, we also present the data showing performance
variety observed across two days by a database component deployed in a real cloud. We have benefited considerably from previous works which show the presence of short-span and long-span performance problems in the cloud. This has motivated us to work on the problem of monitoring enterprise applications at the granularity of user session. We position the design goals of our monitoring framework based on how enterprise applications would be affected by these performance issues.

Many tools like the X-Trace, Dapper, Magpie, ARM study the problem of distributed call tracing [17, 20, 13, 7]. The design of our framework is similar in spirit to that of X-Trace. X-Trace was the first paper to show the advantage of focussing on data path rather than the control path for distributed tracing. They also pioneered in presenting the problem with using common log formats of an IT installation, to follow tasks through hints like IP addresses, username, timestamps. These information cannot be used to represent causal relationship across a task and is not useful when crossing domains. X-Trace showed the need for a user generated UID or TaskID to be propagated along the data path to efficiently trace path taken by a request. In our custom coded solution, we also use a UID to be sent along with the request to correlate measurement data obtained from different components. Whereas in our prototype implementation of the instrumentation framework, we have used X-Trace which helps in generating the TaskID for a user session. This helps us in formalizing the binding data in the form of a metadata which will be of constant size.

Magpie also correlates information available from events generated by the operating system, middleware and application instrumentation like X-Trace. But they focus on a single system or a distributed system extensively instrumented in a highly compatible manner. Dapper is also customized for google’s RPC. Our instrumentation framework provides a simple and scalable solution which is separated from the application by the use of aspects [21]. It can also be easily integrated with any application with minimal development effort.

We differ from X-Trace in many ways. Native X-Trace framework has been designed for providing distributed call tracing information for every task. It requires an application to log every communication end points so that the central server can generate the task graph. Since we require just the time-periods across methods to compute the aggregated component latency and communication latency for every user session, in our implementation we log the time span of a request in every method. This way we save by at least 50% on the number of events generated by native X-Trace.

The central server provided by X-Trace is not scalable [22] due to the bounded buffer. The enterprise applications generate measurement data at random time and frequencies and the X-Trace server is found to be not scalable for a similar use case. In our prototype we make the collection framework scalable by the use of cloud provided infrastructure services like the storage and the queue. This helps in decoupling the log generation and log collection, thereby making the whole infrastructure scale in-line with the application. We also have a log server deployed with the application component like X-Trace, but the purpose of our log server is to aggregate the measurement data obtained from the application. This helps in reducing the number of writes made to the storage. This way the design of our monitoring framework falls closely in-line with X-Trace but abstracts out information that is not relevant to our purpose.

To the best of our knowledge, the properties provided by our monitoring framework like the easy integration with applications, scalability in the cloud, flexible measurements are unique to our framework. In our work, we have also tried to formalize the process of monitoring user observed performance of enterprise applications in the cloud.

7 Conclusions

In this work, we have shown that it is critical for multi-tier enterprise applications to measure and monitor performance of the various components that are deployed in the cloud. We have presented our analysis of the performance characteristics of the various components and services that form a part of our monitoring framework. We have also presented the design decisions and their implications on the application, scalability and feasibility of the monitoring framework.

We have shown the benefits of the design and implementation of our monitoring framework over state-of-art tracing frameworks like X-Trace. We have illustrated the scalability issues with the existing tools and the unique challenges that are presented by the cloud environment. Our evaluation shows the importance of our model and the benefits of abstracting away the details that are not critical for latency measurements. Our results clearly show the following benefits when compared to an off-the-shelf implementation with X-Trace. First, the aggregation made possible by our design reduces the number of writes required to the storage by almost 78% when compared to X-Trace. Second, the accumulation reduces the data overhead of the log measurements shipped by almost 75%. This reduction can result in significant cost benefits given that the framework needs to monitor the application continuously.
Finally, we have illustrated the usability of our tool with the formalization of the requirements as a set of specifications that make the integration of the monitoring tool with the application easier. We have observed that our prototype implementation of the instrumentation framework on a benchmark application shows around 80% reduction in the effort (measured as LOC) required for this integration.

While our framework is a fully functional prototype for enterprise applications deployed on the cloud, we believe that we have only explored the surface of what could be an entire realm of active research. The sensitivity of our design to cloud environments, in particular on the scale and nature of shared deployments are exciting prospects that attract future research.

8 References

[22] W. Wang. End-to-end Tracing in HDFS.