Influence of Architecture Validation on Performance Engineering

Krithika Hegde and Amith Shetty

krithika_hegde@mcafee.com, amith_shetty@mcafee.com

Abstract

It's common for humans to err, so goes with software makers. We don't just come across performance issues they generally are designed into the product. We make reasonable assumptions when we design and write the code and that's where we stumble. With the increasing adoption of virtualization and the transition towards Cloud Computing platforms, modern business information systems are becoming increasingly complex and dynamic. This raises the challenge of guaranteeing system performance and scalability while at the same time ensuring efficient resource usage.

This paper exhibits an approach for analyzing the performance of layered, service-based enterprise architecture models, which comprises of two phases- analysis of workload parameters with 'top-down' propagation, and a 'bottom-up' propagation of performance or cost measures. It also aims at detecting and predicting performance problems at the early stages of development process by evaluating the software design or deployment using simulation, modeling or measurement. Considering the tradeoffs that come with security parameters and their impact on performance. People use terms "performance" and "scalability" as synonyms, this paper concludes with few highlights on how the two are quite different problems having the same symptoms.

Biography

Krithika Hegde is a Senior Software Development Engineer at McAfee, with over 8 years of experience in product development and Integration Activities. She has participated in various product life cycles and been a key contributor to various releases within McAfee. She is passionate about leveraging technology to build innovative and effective software security solutions.

Amith Shetty is a Technical Lead, at McAfee, based in Bangalore, India. He has over 10 years of experience in building applications in Microsoft .NET technology stack. He is experienced working on highly scalable, fault tolerant cloud applications with Amazon Web Services

1 Introduction

Nowadays, performance and scalability requirements have drastically increased to cater modern business information systems. It is often seen that the performance validation is pushed to the end of the release cycle this could result in uncovering problems that can jeopardize the original project goals and deadlines. Hence, it is essential to focus on performance at the early stages of development.

We often see that it is a dedicated (Quality Assurance) QA team's responsibility to execute performance testing. On the contrary, if everyone responsible for the development lifecycle contributes their bit to improve the performance then we can surely ensure that when the output gets to the dedicated performance analysis team it is already in an optimized state.

When it comes to the overall performance of a system these are few questions that cross our mind:

1. How well an application handles the incoming demand?
    a. How can we estimate the average response time?
    b. How large should buffers be?
2. Given a maximum acceptable response time, what is the highest demand the web site can handle?
3. How will you handle unexpected load?
    a. Do you drop additional request or downgrade response time for a higher throughput?
    b. What are parameters do you decide on the elasticity?
4. Which component is the bottleneck?
    a. Should it be upgraded or replicated?
    b. How does the transmission rate of the data affect performance?

2 Differentiating Performance from Scalability

It very important to understand the subtle difference between performance and scalability. In an everyday scenario we often hear people saying, "the performance of the application is bad", but does this mean that response times are too high or that the application cannot scale up to more than some large number of concurrent users? The symptoms might be the same, but these are two quite different problems that describe different characteristics of an application.

Performance refers to the speed and effectiveness of system under a given workload within a given timeframe. We consider several factors when talking about performance efficiency (ISO 25000 Standards, 2018):

- Time-behavior: degree to which the response and processing times and throughput rates of an application, when performing its functions, meet requirements.
- Resource Utilization: degree to which the amounts and types of resources used by an application, when performing its functions, meet requirements.
- Capacity degree: to which the maximum limits of the application, parameter meet requirements.

Scalability on the other hand, is the ability of an application to handle a sudden increase in workload. The increase could come from a surge in simultaneous users, a rising amount of data being processed, or it could also arise from large number of individual requests for access. The increase could be from anything that places higher demand on available resources. Scalability requires applications and platforms to be designed with scaling in mind, such that adding resources results in improving the performance. In case redundancy is introduced, the system performance should not have an adverse effect.

3 Use case

Let us take into consideration a management platform application (Internet of things/Security) which integrates with several third-party plugins/products. In our example let us focus on one workflow of the management platform system. In a big customer environment this application will have the responsibility of managing thousands of end points based on the size of the organization. Each of these nodes would have plugins deployed that can respond back with different types of events. Server involves in parsing these events and aggregating them for further investigation.

In general, management platforms consist of several products with different capability, let us focus on one such work flow of a plugin that is part of the management platform. The workflow for this plugin would begin with the admin analyzing a lot critical information that is fed into the console in the form of event information, system characterization and several other aspects. The admin then will create tasks to search for suspicious behavior and characteristics in the end points in the customer environment. This search operation triggered by the user needs to be run on each of the end nodes. The data returned by the search operation from several thousand nodes needs to be stored for a short duration and provided to the user on his console.

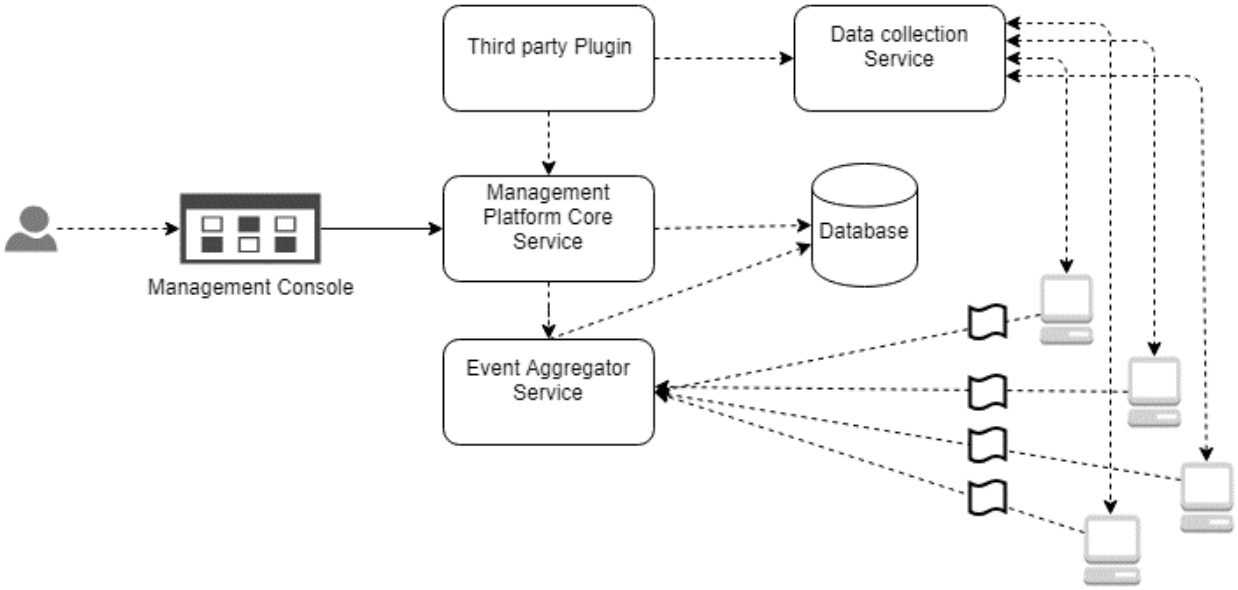

Figure 1 Management platform plugin workflow

Before we dive into the details of the above system it is important to understand performance expectation from the perspectives of different stakeholders

4 Perspectives on performance architecture

The views on architecture are aimed at different stakeholders that have an interest in the modelled system. Also, for the performance aspects of a system, many perspectives can be discerned, resulting in different (but related) performance measures

End User perspective: The user of the system would be interested in response time, which is the time between issuing a request and receiving the result. The response time is the sum of the processing time

and waiting times taking into consideration the synchronization loses. In our workflow this refers to the time required to retrieve search results from each of the end node.

Process perspective: the performance consideration that need to be taken from the process perspective. It refers to the time required to complete one instance of a process which could involve multiple products, services etc., as opposed to the response time, which is defined as the time to complete one request. This perspective takes into consideration the entire process of input events arriving at the system, aggregation of these events made by other plugins and the think time the user puts in before triggering the search operation.

Product perspective: As the above example consists of platform management system with several plugins/products we would need to consider the processing time for a product for a given workflow, the amount of time that actual work is performed for a certain product or outcome. In this scenario the response time is considered without waiting times. Hence it can be orders of magnitude lower than the response time.

System perspective: This perspective focuses on the performance consideration for a given system in the workflow. It can be considered as the throughput, the number of transactions/requests that are completed per time unit.

Resource perspective: It is very critical to understand how well resources can be utilized, this refers to the percentage of the operational time that a resource is busy. In terms of performance utilization can be considered a measure for the effectiveness with which a resource is used. On the other hand, a high utilization can also be an indication of the fact that the resource is a potential bottleneck.

5 Analysis of architecture components

Now that we have a good understanding of the workflow and performance expectations, we can take a deeper look at service layers and implementation layers. By identifying common architectural implementation errors, many potential performance issues can be caught and fixed early in the development process, including excessive remote service calls or making too many database retrievals, which can introduce architectural problems like latency bottlenecks.

5.1 Detecting performance problems at design phase

Let’s have closer look at the layers present in the current workflow. A service layer exposes external functionality that can be used by other layers, while an implementation layer models the implementation of services in a service layer. Thus, we can separate the externally observable behavior (expressed as services) from the complex internal organization (contained within the implementation layers). The implementation layer further relies on the infrastructure layer for serving the request that comes from the Service layers, like connections for database or network related operation between plugins.

In a layered view, analysis across layers is possible by propagating quantities through the layers. Which is nothing but the arrival frequency of the requests that come from the user. To analyze the load propagation between layers we need to first characterize our workload. This characterization can be done based on whether the load propagated through the layers is rhythmic and regular, whether it varies seasonally or by time of day, whether it is growing over time, and whether it is inherently subject to potentially disruptive bursts of activity. These workload characteristics must be understood for performance requirements to be properly formulated and for the system to be architected in a cost-effective manner to meet performance and functional needs (Andre 2008). The main components of the workload are identified by the amount of traffic that arrives at the system from an external source. Some examples include HTTP requests, Web service invocations, database transactions, and batch jobs.

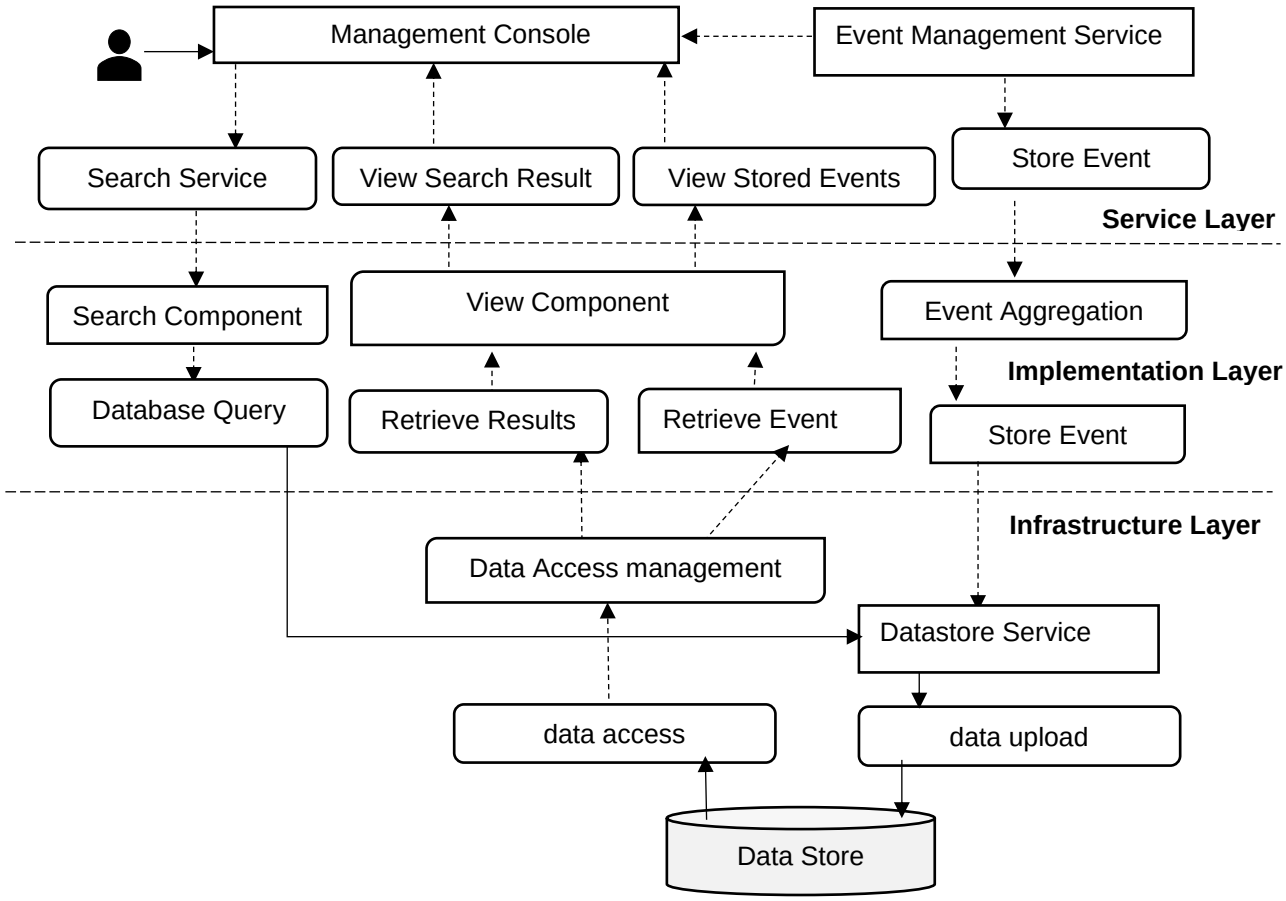

Figure 2 Layered software architecture

In our flow we can consider the arrival frequency of requests as the “demand” from the topmost layer which consists of the admin/ users. These requests are propagated towards the deeper layers of the architecture, yielding the demands of each of the model elements. Once the workloads have been determined, we can determine the effort these workloads require from the system resources. This effort can be expressed in terms of performance indicators: Throughput, System response time, Queue time for events, and Queue length for components. The throughput and the system response time indicate the overall performance of the system, while the other two metrics provide information about the internal behavior of the system, e.g., contention, possible performance bottlenecks, and starvation.

From the ‘deepest’ layers of the models, these measures are propagated to the higher layers. In general terms we can assess the performance of a system by the degree to which the system meets its objectives for timeliness and the efficiency with which it achieves this. We can measure timeliness in terms of meeting certain response time and/or throughput requirements, response time referring to the time required to respond to a user request (e.g., a Web service call or a database transaction), and throughput referring to the number of requests or jobs processed per unit of time.

We can use the throughput and response time values to create a performance profile for top down flow and bottom up flow of workload. Using this we can figure out how queueing times from the lower layers of the architecture accumulate in the higher layers, which results in response times that are orders of magnitude greater than the expected results which are derived during the profiling.

In the workflow mentioned above Events are analyzed by the user and the query to search for similar behavior on other end nodes is triggered by the user. As the system needs to trigger this search on all the

systems in the environment, this search task will be queued to be executed on the end nodes. If the queuing time from the search task goes over the permissible utilization threshold then the response time to retrieve search result task is going increase drastically.

Below is the formula derived from Little's Law (Simchi and Trick. 2013) for application in Performance analysis scenarios.

Arrival rate or Throughput(λ):

$$\lambda = L/R \quad (1)$$

L- no of requests/users in the system R- Response time

Another useful relationship is the Utilization Law, which states that utilization is throughput times service time. We can derive **Utilization(ρ)**

$$\rho = \lambda S \quad (2)$$

If you have M number of servers,

$$\rho = \lambda S / M \quad (3)$$

λ- Arrival Rate S- service time

We also consider the response time which consist of the wait time and the service time can be donated as **Response time(R)**

$$R = W_q + S \quad (4)$$

The response time in an M/M/1 queuing system:

$$R = S/1 - \rho \quad (5)$$

W_q -Wait Time S - Service time

Analysis of workload from top layers to the bottom layers:

We have taken a small subset of values from our experiment to depict performance evaluation on the model depicted in Figure 1. Below table shows the workload for services 's' in the Figure 1 in terms of arrival rate λ. The arrival rate depends on the frequency of requests coming from the administrators/users. We have scaled the arrival rates as arrivals per second.

Analysis of performance from the bottom layers to the top layers:

This shows the performance results i.e., the processing and response times for services and utilizations for resources at the implementation and infrastructure layer. For simplicity we have assumed Poisson arrivals (This means that the requests arrive randomly and independently of each other, with average rate λ) and the time between arrivals is exponentially distributed, with mean 1/λ.

Resource(r)	Service(s)	λ (sec^{-1})	S (sec)	R (sec)	U
Data Store	Data access	0.0382	6.0	7.8	0.229
Data Store	Data upload	0.0278	0.2	0.2	0.006
Data access Component	Retrieve results	0.0313	12.8	25.0	0.488
Data access Component	Store events	0.0069	12.8	25.0	0.488
Datastore Component	DB query	0.0278	0.7	0.7	0.019
Datastore Component	DB update	0.0069	0.7	0.7	0.019
Search Component	Search results	0.0278	1.2	1.2	0.025
View component	View result	0.0313	27.0	172	0.813
Event aggregator	Event aggr.	0.0069	33.7	44.0	0.234

Table 1

The results show that the queuing times from the lower layers accumulate in the higher layers. The 'view' component of the platform management application has a utilization of over 81%, which results in a response time of the 'view search result' service of almost 3 minutes.

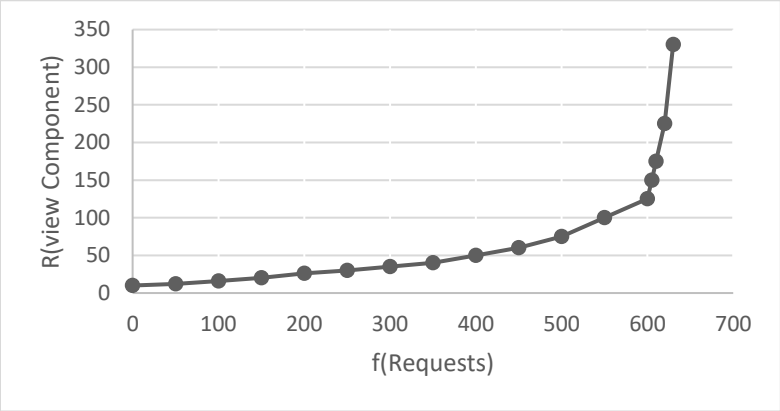

Figure 3 Arrival rate vs. response time

In the design stage these results help us decide the number of components required during the actual development phase.

There has been a lot of research in creating automated models and calculating performance at early stage of development (Balsamo,Marco, Inveradi and Simeoni. 2004, 295-310)

- Model-based software performance analysis introduces performance concerns in the scope of software modeling, thus allowing the developer to carry on performance analysis throughout the software lifecycle. (Cortellessa, Marco and Inverardi 2011)
- Evaluation of non-functional properties of a design (such as performance, dependability, security, etc.) can be enabled by design annotations specific to the property to be evaluated. Performance properties, for instance, can be annotated on UML designs by using the "UML Profile for Schedulability, Performance and Time (SPT)" (Bernardi, Donatelli and Merseguer. 2002,35-45). However, the communication between the design description in UML and the tools used for non-functional properties evaluation requires support, particularly for performance where there are many alternative performance analysis tools that might be applied. (Woodside,Petriu, Shen ,Israr and Merseguer. 2005)

As a good performance engineering process, we recommend predicting (at early phases of the life cycle) and evaluating (at the end) performance of an application, based on performance models, whether the software system satisfies the user performance goals.

5.2 Architecture validation during development phase

In section 5.1 of this paper we saw how we could analyze performance of an application at design phase. We would like to talk about an automated solution to evaluate performance that can be deployed in our development cycles, which could help accelerate the overall product performance. As part of this automated solution we add checkpoints that we consider are crucial in creating a feedback loop based on the inputs obtained during the runtime monitoring of system performance. We are all familiar with concept design patterns which defines good practices to design software. Conversely, the concept of antipatterns has been defined for characterizing bad design patterns. In this context, Smith and Williams introduced performance antipatterns (Smith and Williams 2002), which are bad design practices that may lead to performance degrade. We would like to state some common performance design flaws and how adding a rule check for each these performance antipatterns helps in creating a feedback loop to the software architectural models.

5.2.1 Scattered Information:

The information required by an object is scattered in several different places. The processing required to retrieve this information from different sources results in application suffering from performance problems. In our work flow as depicted in figure-1 the user retrieves information from the events table and the aggregation plugin then uses those results to query the table which has the results of search performed on the end nodes. The impact on performance is huge when the data is on remote server and each access requires transmitting all the intermediate queries and their results through the network. The above stated performance problem can be tied to the famous performance antipattern **Circuitous Treasure Hunt** (Smith and Williams. 2000)

In large customer environments we deal with huge result sets, one object invokes an operation on another object, that object in turn invokes an operation on another object, and so on, until the desired result is obtained. Then, each operation returns, one by one, to the object that made the original call. The performance of the systems will deteriorate due to such chaining of the method invocations among different objects. This chaining of the method invocations at different objects requires an extra processing to identify the final operation to be called and invoked, especially in distributed object systems where objects involved may reside in other processes and on other processors. The impact on performance degradation will be even greater, when every method invocation causes the intermediate objects to be created and destroyed. Allocation and De-allocation of memory on intermediate objects are performed frequently and consumes system resources.

Rule check:

- Inspect if the software entity is retrieving a lot of information from a database.
- Check if the software instance generates many database calls by performing several queries
- The processing node on which the database is deployed might show high utilization value for CPU(s) and disk(s). A check needs to be added to capture rise of at least one of these values above the performance threshold boundary.

Generally, a database transaction usually requires a higher utilization disk device instead of CPU, hence the max value for CPU is expected to be larger than the max value for disk.

5.2.2 Overusing database server for running code

In an ideal scenario it is more efficient to perform processing actions close to the data, rather than transmitting the data to a client application for processing.
In our workflow as depicted in Figure 1 we noticed that performing parsing of events and several other operations on the database end resulted in database server spending a lot of time in processing rather fetching data and serving client requests. As the database is a shared resource for both the event service and the search service it can become a bottleneck during high use. If in a cloud environment where Database Transaction units are charged the runtime costs would be huge as the datastore would be metered.
Moreover, moving the processing into computing resource which are scalable horizontally would always be better.

By doing so we should take into consideration of not moving the processing, if doing so can cause the database to transfer far more data over the network.

Rule check:

1. Add a check to monitor database activity, make sure to add rules that run a mixture of suspected scenarios with a variable user load. Telemetry data can be used to support this test.
2. Check the source code for DB activity that reveals significant processing but very little data traffic.

5.2.3 An excessive number of requests is required to perform a task

Small amount of information is sent in each message. The amount of processing overhead is the same regardless of the size of the message. With smaller messages, this processing is required many more times than necessary

Rule check:

Consider a software entity that exchanges a huge number of messages with remote entities.

1. Check for inefficient use of bandwidth, by checking if the software entity under inspection is sending high number of messages without optimizing the network capacity. The process node on which this software entity is deployed needs to be checked for utilization of the network lower than the given performance threshold.

5.2.4 Stuffing of the Session

User session during the web site visit can be tracked using the Session object. It generally starts with putting very little information in the session, but over a period the session object keeps on growing. At times unnecessary/redundant information is stuffed into the session object. The objects placed in the session will last till the session object is destroyed.
This impacts the number of users that can be served by the application at that instance.

Rule check:

1. Get details of what goes into the session object and filter out unnecessary information
2. Track objects that can be removed from the session when the usage is over
3. Check for objects that can be defaulted to request scope.

The above mentioned antipatterns are frequent causes of application performance issues. The teams usually start with the right intentions but over a period, things will start slipping. Some of the common reasons

5.3 Role of concurrency models on Performance

Performance throughput can be improved to a great extent by appropriately designing various concurrency models such as thread pool, thread-per-connection, or thread-per-request. The design should be efficient enough for threads to be able to process new incoming client requests even while other threads are blocked waiting to receive a response from a backend server. Due to the cost of thread creation, context switching, synchronization, and data movement, however, it may not be scalable to have many threads in the server.

Each of the above-mentioned concurrency models have the following limitations:

Thread pool—In a thread pool model, the number of threads in the pool limits the throughput. For example, if all threads are blocked waiting for replies from backend servers no new requests can be handled, which can degrade the throughput of busy middle-tier servers.

Thread-per-request— In this scenario a thread will be created for every incoming request from a client. Server can create a number of threads as per request. This concurrency model may not scale when a high volume of requests spawns an excessive number of threads.

Thread-per-connection— is the concept of reusing the same HTTP Connection from multiple requests. This model may result in server running out of threads if many clients connect to the server at once. If the server is slow in processing a client request, a single client may create many connections and threads on the server, further slowing it down. Moreover, if a server is busy processing a client's request, that client can open a new connection to send a new request. Also, the server has no way of knowing when a given (persistent) connection can be closed. If the browser doesn't close it, it could "linger", tying down the Thread Per Connection thread until the server times out the connection

Let us further consider the problem with thread per connection

- **CPU** – Thread switching is considered very expensive. The OS may switch between threads at any time and needs to save the entire thread state so that it can restore it when it switches back. For 10,000 threads, the overhead of switching takes up more than 40% of the CPU resources.
- **Memory** – Each thread would need a stack for all its function parameters, return values and local variables, which is large enough to prevent a stack overflow. If we consider a standard stack size of 1MB, for example, we will be dealing with 10GB of memory for 10K connections.
- **I/O** – We see that most of the server activity is I/O, like sending a request to another microservice or calling a database server. Therefore, at any given time, most of the threads will be idle, waiting for some I/O to return.

How the use of event loops helps dealing with above issues

In today's market, we have access to many technologies which have implemented a paradigm known as event loops. These servers use a single-thread which runs through a loop of CPU-bound code, using callbacks to manage I/O completion. In these cases, I/O is implemented using the operating system's asynchronous APIs. This method deals with all the drawbacks previously discussed:

5.4 Extraction of architectural performance models from execution traces

Real-world execution traces collected can record performance problems that are likely perceived at deployment sites and usually triggered by complex runtime environments and real-world usage scenarios. By analyzing large-scale and diverse execution traces collected from deployment-site computers, performance analysts and developers can gain useful knowledge for system design and optimization

However, finding performance problems from real-world traces is challenging. The problems can be rooted subtly and deeply into system layers or other components far from the place where delays are initially

observed. Performance analysts should spend great efforts to confirm the existence of subtle problems, and to figure out how they were caused and spread. Given many real-world execution traces where bad performance may be amortized over various underlying problems, such performance-analysis activities become harder or even infeasible.

To tackle challenges of identifying deeply rooted problems, we propose a new trace-based approach consisting of two steps: impact analysis and causality analysis. The impact analysis measures performance impacts on a component basis, and the causality analysis discovers patterns of runtime behaviors that are likely to cause the measured impacts. The discovered patterns can help performance analysts quickly identify root causes of perceived performance problems.

6 Implementing architecture validation into build process

The goal of performing architectural validation as mentioned in section 5 is to recognize potential trouble spots in the application runtime behavior that can lead to performance and scalability problems. To achieve this, we would need a solid set of architectural rules to be defined by the software architects.

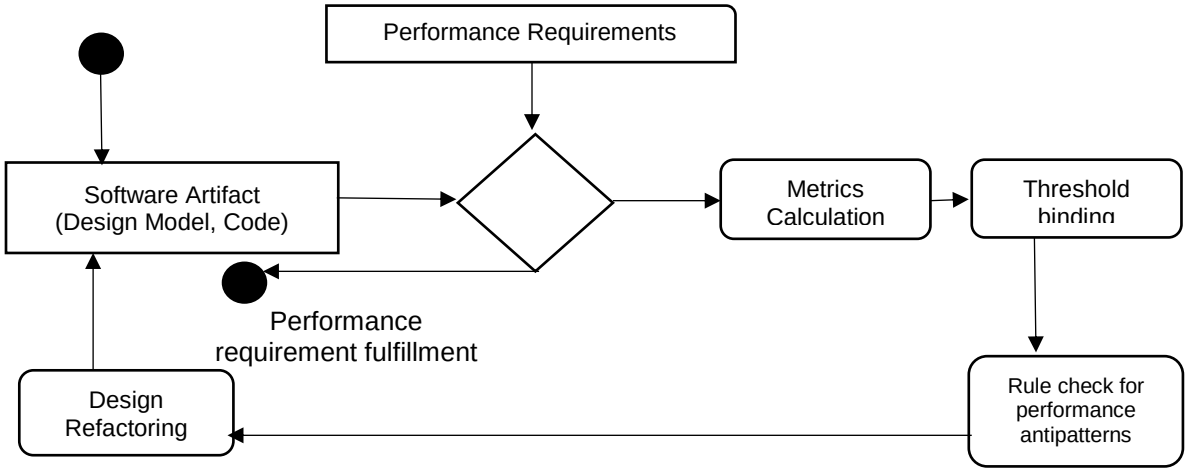

Figure 4: Performance architecture validation cycle

Our Proposed approach starts with the design model or existing version of the application, that does not fulfill certain performance requirements like the response time for a given service. To make sure the performance goals are met for the application it is required to conduct performance analysis. For an application that is in design phase, we can incorporate performance considerations using performance modelling techniques and for the application in development phase we can make use of performance monitoring methodologies to aid in improving the performance of the application. Based on the requirements gathered and previous programming experience we can add architecture rule checks to find the performance bottlenecks at an early stage and provide valuable feedback to avoid performance issues at later stages of the development cycle.

As part of the agile development methodology, it is required to test using stable system on regular and continuous intervals. Therefore, we would need to trigger automated architecture rule check framework within the build process. The results as with functional tests can be formatted as Junit or HTML reports. In case of execution errors we could use logs or tracing data. In addition to the results obtained from functional tests, performance-test results are analyzed and recorded for each build, and it is this integration of

continuous testing with the agile ideal of continuous improvement that makes this method so useful. Developers can collect feedback on specific components in a sort of continuous feedback loop, which enables them to react early to any detected performance or scalability problems.

Choosing the apt tracing and diagnostic tools to measure performance and to determine architectural metrics can be substantially hard. The biggest problem with the current profiling tools, from the most basic to the most sophisticated is that they do not perform any meaningful analysis on the data collected. The most sophisticated tools merely perform statistical analysis on performance metrics and give average resources consumed by the different components/classes. When profiling large enterprise applications, the amount of information recorded can be truly overwhelming. It can be difficult and time consuming for developers that have to analyze the data produced by these tools in search of particular problems. Furthermore, even when developers manage to identify issues often they are unsure as to how to go about the issue.

Another problem with the current tools is that they tend to focus on identifying low level performance bugs that exist in the system. The tools, however, do not focus on analyzing the system from a design perspective. Thus, design flaws can easily go unnoticed. While identifying and fixing low level bugs in the system will very often improve the system performance, it is common that this is not enough to meet performance requirements.

To be able to deduce the run-time design from an application we need to identify the relationships that exist between the different components that make up the system. These relationships can be captured by recording run-time paths. There are commercial application monitoring tools, such as DynaTrace. Which uses its own PurePath technology which captures timing and context for all transactions across all application tiers. It has support for both Java and .NET environments. These paths maintain the order of calls between components they also capture communication patterns between the components. Such communication patterns can be analyzed to identify inefficient communication between components. We can also deduce performance metrics (such as CPU and memory usage) on the component methods. Below is a sample architecture validation rule snippet available through Dynatrace

```
▼<archival>
  ▼<rules rulesDashboard="RulesDashboard">
     <rule name="OnePurePath" description="Verifies that there is at least one PurePath" dataset="//purepaths
     <rule name="3APIs" description="Verifies that there are 3 APIs" dataset="//apis/api" verifyType="null" e
     <rule name="10MethodCalls" description="Verifies that the sum of all method call counts is not greater t
     source="count"/>
     <rule name="CallsHandleInternal" description="Verifies that there is a method call to handleInternal()"
     source="method"/>
   </rules>
  ▼<transactions transactionDashboard="TransactionDashboard">
     <transaction transactionDashboard="TransactionDashboard" pattern="*" rules="OnePurePath,3APIs"/>
     <transaction pattern="test.*" rules="CallsHandleInternal"/>
     <transaction pattern=".*NA=.*" rules="10MethodCalls"/>
     <transaction name="ByTraceName?testHomeHandler" rules="*"/>
   </transactions>
 </archival>
```

Figure 5 Architecture validation rule checks using Dynatrace

Tracing and Monitoring tools like Dynatrace have also enabled REST API which allows us to programmatically query for all the required information using Java.

In our management platform workflow, we encountered a major performance mishap, due to one of the third-party plugin which had few design flaws. Due to which the entire management platform was failing to function. To avoid such mishaps, we worked on coming up with a performance rule check framework with relevant architecture validation rules in place. The rule check framework would analyze a plugin deployed on the management platform and would monitor its interaction with the database and several other core

services of the management platform. As this rule check helped maintain the overall performance of the platform, we would recommend this mechanism of including this architecture validation process as part of the performance engineering process.

Below is a graph with few response values extrapolated for the search result component that we have seen in figure 2. As seen in the below graph we see the difference in the response time of the search component has improved in the consecutive builds after incorporating few design changes based on the rule checks.

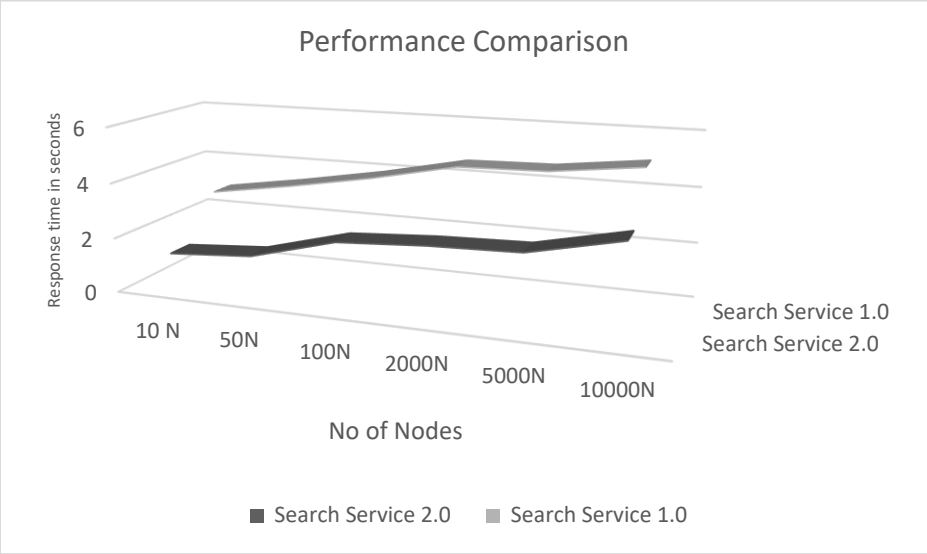

Figure 6 Performance Comparison of search implementation between 2 different builds

7 Conclusion:

In the grand scheme of things, you would need to take one of the three options to increase any given application's performance and scalability:

- Invest time in improving the performance of the application right from the initial stages of the development lifecycle
- Migrating the application or individual tiers to nodes with greater capacity- which is also known as Vertical Scaling.
- Instead of adding more resources, which is in the case of vertical scaling, a web application or its tiers are decoupled so that demand is spread out among multiple nodes- which is also known as horizontal scaling.

Which of the above three approaches should be considered depends on a series of factors, including the specifics of you web application, expertise of the team, initial design considerations of the application and what is more attainable given the resources you have in hand.

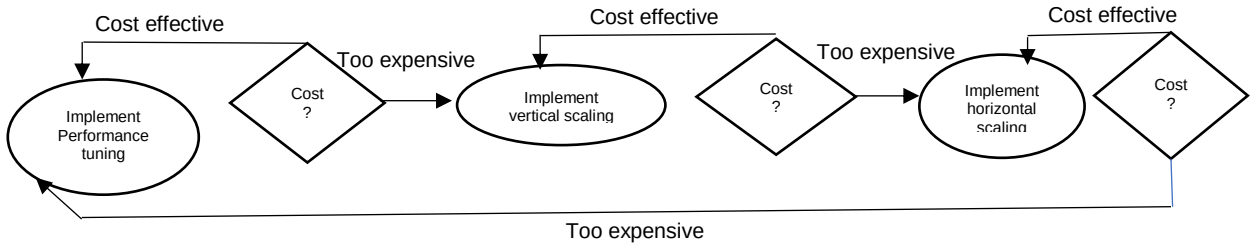

Figure 7 decision tree for performance implementation

As depicting in the above figure, what factor is too expensive totally depends on the current state of the application. If the decision is made to implement performance tuning, it is very much required to a have well experienced team which can incorporate several checks right from the design phase, if that is not feasible it can be easier to skip to the next step of vertically scaling an application or vertically scaling its different tiers. By the same notion, if the service providers or data centers are unable to provision vertical scaling then it would be easier to skip to the step of implementing horizontal scaling on the application tiers or horizontally scaling tiers in themselves. If neither of the scaling options are possible and if there is an experienced team, sticking to performance tuning using the approaches mentioned in this paper maybe a better alternative.

8 References

Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M. 2004: Model-based performance prediction in software development: a survey. IEEE Trans. Softw. Eng.

Bernardi, S., Donatelli, S., Merseguer, J. 2002: From uml sequence diagrams and statecharts to analysable petrinet models. In: WOSP, pp. 35–45

Cortellessa, V., Di Marco, A., Inverardi, P. 2011: Model-Based Software Performance Analysis. Springer, Berlin

Woodside, C.M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J. 2005: Performance by unified model analysis (PUMA). In: WOSP, pp. 1–12

C. U. Smith, L. G. Williams.2000 “Software performance antipatterns”. WOSP.

Simchi-Levi, D.; Trick, M. A. (2013). "Introduction to "Little's Law as Viewed on Its 50th Anniversary"". Operations Research.

ISO 25000 Standards. (2018). Retrieved from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/59-performance-efficiency