SADIO Electronic Journal of Informatics and Operations Research

http://www.dc.uba.ar/sadio/ejs

vol. 11, no. 1, pp. 49-65 (2012)

Managing Scalability, Availability, and Performance Requirements for Cloud Services

Rodolfo Kohn[1]

[1] Argentina Software Design Center
Intel Corporation.
Corrientes 161
5000 Córdoba, Córdoba, Argentina
e-mail: rodolfo.kohn@intel.com
No. tel. 54-351-5265646

Abstract

When developing services software meant to reach millions of users through the Internet, unlimited scalability, high performance, and 100% availability turn essential qualities that will support, or seriously disrupt, business growth. This is probably the most important challenge for the new software and infrastructure paradigms in the cloud. This paper proposes a combination of measurable and non-measurable requirements, specific, usually overlooked, process activities, assets, and guidelines to manage these software qualities from the early product conception and then throughout the whole product lifecycle. It discusses difficulties found incorporating these practices into the software development process in real-world projects as well as their solutions. Furthermore, concrete results and lessons learnt are presented showing advantages and drawbacks of the approach.

Keywords: Scalability, Performance, Availability, Requirements, Cloud, Services, Non-functional Requirements, Software Qualities

1 Introduction

1.1 Problem Statement

If functional requirements define system behavior, non-functional requirements (NFR's), a.k.a. qualities, determine how well the system behaves. Nonetheless, it is difficult, if not impossible, to specify non-functional requirements in a SMART way, to track them, to arrive to the best solution, and to assure they are satisfied. For this reason, developers and managers tend to push their consideration as late as possible in the development cycle. As a consequence, often problems are discovered too late, in production when customers have already been affected, or, at best, during system test a few days before planned release. Usually, NFR's are treated as soft goals, i.e., goals that need to be addressed not absolutely but in a good-enough sense [Chung, 2009] and probably this is what makes them so difficult to manage.

Performance, scalability, and availability are challenging system qualities to drive during the software development process and even more during the whole product lifecycle.

This is due to various factors including the difficulties inherent to all non-functional requirements. First, it is usually impossible to specify these requirements in an absolute and measurable way [Chung, 2009]. Second, there is high level of subjectivity when evaluating these qualities. Third, it is always possible to improve performance and scalability for a certain cost: there is a continuous trade-off between good-enough and cost. Fourth, it is expensive to validate these requirements due to a tight relationship with the infrastructure and the large spectrum of workload combinations. Fifth, it is impossible to validate these requirements in an absolute way: you just reduce risk margins.

Probably for lack of experience, it is difficult to convince managers to invest resources to avoid future problems while weighing against giving up required functionality. The same occurs with the developers pressed to implement more functionalities in a shorter time. Finally, performance and scalability acceptance criteria usually are not adopted for approving a release.

However, new business models brought by the Internet, technology progress, and hardware commoditization have changed the panorama.

On one side, Internet enables the possibility to reach millions of users on a 7x24 basis with no need to invest in immense capital. Whereas this is wonderful for business, it is a real challenge for software engineers and IT administrators. Indeed, software products have to endure greater, and highly variable, workloads. Availability turns essential to keep business growing. Software will undergo stringent conditions that will bring out every "unexpected" and hidden error. On the other side, technology advance has made hardware more accessible thus allowing small and medium companies to buy devices like load balancers, to adopt disaster recovery solutions, or to use Infrastructure-as-a-Service and Platform-as-a-Service, such as provided by Amazon Web Services [Amazon, 2012], Rackspace [Rackspace, 2012], Google Apps Engine [Google, 2012], Windows Azure [Windows Azure, 2012], etc. As a consequence, there is no economical reason for enterprise software not to adopt scalability and high availability concepts and not to exploit their benefits. For software engineers this implies new architectural paradigms and the fact that software issues will have greater impact.

In this new world of cloud services, performance, scalability, and availability turned essential for business success:

- When reaching millions of users across different regions, systems have to deal with unpredictable workloads. Even though a system may run with a light workload at the beginning, if business is successful a dramatic peak may suddenly occur. Each service may have a different rate of adoption, and sometimes there will be little advance ramp-up before an exponential spike in demand. Systems must be able to adapt and cope with the new load transparently for end users.
- In a competitive and brave world, if users have problems accessing a web site or a service, either for unavailability or for unacceptable response time, they immediately move to a different provider. Not

to mention thousands of blogs, newspapers, and rest of the media, eager to wreak havoc when well known companies are involved. In a research commissioned by Compuware [Compuware, 2009], after 1,538 interviews with Internet consumers, it was found out that:

- 51% of them spent most of their retail budget during peak traffic times, in promotions for instance.
- After a poor user experience with a web site during peak traffic times, 78% of them said they went to a competitor's site and 88% of them were less likely to return to a web site.

- Additionally systems are going to suffer all kinds of denial-of-service attacks and they must be able to deal with them without affecting availability.

1.2 Proposed Solution

Dealing with nonfunctional requirements is not a new problem and various studies proposed distinct approaches to handle them. For example, [Mylopoulos, 1992] proposes a process-oriented approach to achieve non-functional goals. In [Bennett, 2004] a performance engineering methodology based on UML is proposed. However, these works take assumptions that in practice are not always available. For example, in [Mylopoulos, 1992], an input for the process is the performance goals and most of the times defining performance goals is the real problem: usually nobody has defined them and, to do it, teams have difficult discussions to reach an agreement. Engineering teams are reluctant to commit to performance goals for a system that does not exist yet and they do not have a clue about how it is going to behave. Disagreements are revived every time goals are not achieved in performance test. The solution proposed here does not compete with the mentioned methodologies but it consists of a higher level pragmatic approach intending to influence the software development process from the very beginning and throughout the whole product lifecycle. It involves a holistic approach that is not limited to just measurable requirements but also to specifications that are not measurable and guide development and validation phases. It also includes a set of process activities and assets that help achieve these non-measurable requirements by assuring the team is continuously considering the performance and scalability needs throughout the development process and when the application is running in a production environment. Moreover, architectural, coding, and testing best practices are included as part of this solution. This approach was adopted for our projects and it is called SCAP process. SCAP stands for Scalability, Availability, and Performance.

This process may encompass methodologies as the ones referred above and any performance engineering practice.

1.3 Solution Applicability Scope

The solution proposed in this work especially applies to software services that are consumed on-demand and are required to undergo variable demand which oftentimes generates load peaks of unpredictable magnitude.

This type of demand is typically generated by a number of end-users in the order of millions, tens of millions, or more. An important characteristic is the high cost of failures and unavailability for the business. Most users give up after failing accessing the system [Compuware, 2009]. This is in opposition to enterprise software where the workloads are highly predictable and users, typically employees, do not have other options besides those provided by the IT group.

Some examples of this type of software are an applications store, an on-line marketplace, an on-demand CRM application, etc.

This work accepts the cloud definition proposed by NIST in [NIST Definition of Cloud Computing, 2011] and refers to software that typically runs on an Infrastructure-as-a-Service stack or a Platform-as-a-Service stack with any provider and with any infrastructure and platform stack like OpenStack [OpenStack, 2012], for

instance. This software might fall in the category of Software-as-a-Service or even Platform-as-a-Service as the line between them is not solid. The application could run in a private, public, or hybrid cloud indistinctly.

The main premise that motivates this work is that if the software is not designed for the cloud, as defined in [NIST Definition of Cloud Computing, 2011], running in the cloud brings no advantage other than lower cost in a few cases.

2 SCAP Process

SCAP Process created for our projects requires the following activities:

- Requirements definition:
  - Workload estimations at Requirements Gathering phase and more detailed estimations after architecture and design definition.
  - Definition of business performance targets.
  - Identification of business entities and business events and the order of magnitude they need to scale. This task allows software architects to take better decisions in the trade-off of different quality attributes and it provides important input to decide about the data models required. For example, for events processing requiring high scalability a NoSQL database could be the best option.
  - Consideration of special scalability requirements like geo-scalability in different availability zones; content distribution possibly through a content delivery network; disaster recovery; or active-active configuration.
  - Identification of level of availability required.
- Architecture definition:
  - Scalability and performance review.
  - Definition of implementation performance targets stemmed from defined architecture and business performance targets.
  - Estimation of total cost of ownership considering the architecture and workload estimation.
- Performance tests:
  - Performance, scalability, stress, and longevity tests after certain development sprints and before every release.
  - Creation of performance baselines.
  - Performance bottleneck analysis when performance targets are not reached, when system does not scale as expected –scalability is linear if there is no bottleneck- or when performance needs to be improved for any reason.
- Capacity planning based on performance test results, infrastructure utilization counters, IT experience, and business forecasts.
- Before delivering software to the hosting environment, a report per application is generated characterizing it: services supported, transactions per application, detailing components involved per transaction, classification and estimated number of SQL transactions in the database, messages sent back and forth per transaction, estimated HTTP payload length in Bytes per message, expected workload (average, normal spikes, and eventual bursts)
- Monitoring plan in production: internal monitoring -controlling measurements against thresholds and gathering workloads feedback- and external monitoring for determining overall availability and response time.
- Marketing communications plan: this is necessary in order to inform operations, support, and engineering teams of upcoming promotions or other special events that could generate load peaks.

The different activities are shown in Figure 1 in a cascade development model. However, all these activity perfectly fit into development sprints in agile methodologies.

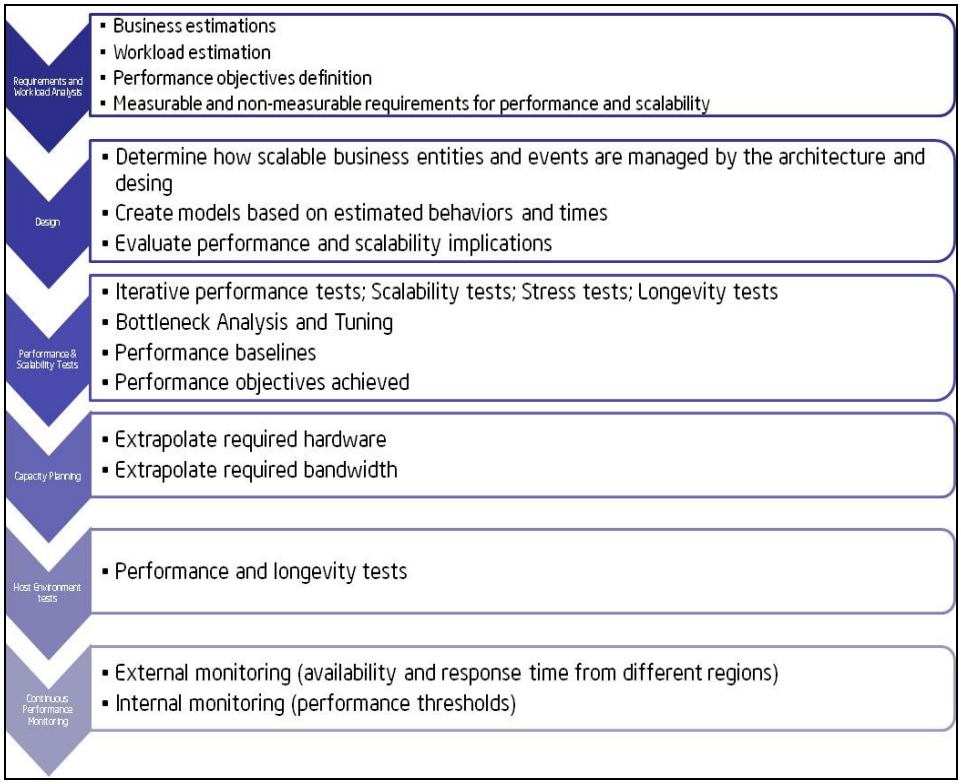

Fig. 1. SCAP process activities.

3 Requirements

It is impossible to specify measurable requirements that will assure performance, scalability, and availability by themselves. However, by implementing certain requirements it is possible to reduce risks of failing in providing the product with these qualities in a good enough level.

It might be argued that certain requirements go too far into design decisions. However, it is also true that if software is not architected and designed according to certain good practices, they will not be able to have unlimited scalability, performance will be compromised under load peaks, and availability goals will be in serious risk.

3.1 Scalability Requirements

Measurable scalability requirements should include the following list:

- The system shall be able to horizontally scale in all layers including data storage [Michael, 2007].
- All software components in all layers shall be replicable with different instances running in different servers, either virtual of physical.
- The system shall be able to horizontally scale both by adding or removing boxes. It shall not fail upon replication or when a replicated instance is removed.
- Given a scale unit formed by a set of servers at all layers. Adding a second scale unit shall yield a minimum of 70% performance increase. This is just an industry rule of thumb and could be higher. If performance had little increment, say 10%, by adding a new server, the system is considered to have

poor scalability, probably due to a poor architecture or design. In general, by duplicating capacity, maximum throughput should be duplicated or, expressed in terms of scale units, a new scale unit must add a throughput equivalent to the first one alone without affecting response time. If this does not happen, it is due to a bottleneck becoming relevant for that workload. A bottleneck analysis should always be required in order to identify the next hinder to scalability.

3.2 Performance Requirements

Measurable performance requirements should be stated as performance targets. Performance targets can be specified as:

- Response time (e.g. end-to-end turnaround time).
- Throughput (e.g. requests per second, transactions per second, etc.).
- Resource utilization (e.g. processor utilization, power consumption, disk access, network utilization).

For backend systems, at least response time targets and throughput targets should be defined. The question that immediately arises is how to define the correct performance targets. This can be a contentious issue and especially when performance targets are not achieved.

Response time targets do not change frequently unless new features are required and this could imply a modification.

Throughput requirements usually are not fixed as they continuously change based on business progress. SCAP requires carrying out a workload analysis and estimation activity to set throughput targets based on business estimations and system architecture. It is also necessary to review them with certain periodicity, e.g. quarterly.

Throughput target specification should follow the following steps:

- Identify scalable business entities: it could be users, units of products delivered (applications, cars, etc.), providers, etc.
- Identify scalable business events: purchases, subscriptions, deposits, etc.
- Obtain forecasts of business entities and events: it is desirable to obtain them from business groups but if it is not possible a best guess should be done based on available data. It should have consensus among all stakeholders.
- Estimate behaviors that will trigger business events and determine average of event occurrence per unit of time. For example, 50% of users purchase on weekends. Marketing professionals' input would be highly valuable in this task. Consensus of all stakeholders is essential.
- It is a rule in IT that systems must not be planned for average but for peak load. A rule of thumb tells that plans have to be done for average multiplied by 4. Alternatively, another heuristics can be applied, using Poisson distribution for example. In any case it is recommended to keep simple rules and automated to make the process repetitive; otherwise it will be too expensive. This is necessary to keep a base capacity. Moreover, it will help determine how to scale out for burst in demand.
- Finally, throughput targets per request, transaction, or operation, are derived from the rules applied.
- These steps should be repeated with certain periodicity, e.g. quarterly.

In the case of response time, other rules apply because it depends on user experience. The goal is to deliver excellent user experience that should not change under load peaks. If the system scales ideally, response time is constant under any load. Usually, response time targets are more stable than throughput.

3.3 Logging and Monitoring Requirements

Logging is especially important for availability. When an error arises, it is essential to detect it fast and also find out the root cause fast. Logs must keep a standard structure, like CIM [CIM, 2011] or ITU X.733 [System Management: Alarm Reporting Function, 1992], in order to enable its automatic processing for detecting errors in real time, root cause analysis, and for load trends analysis.

Monitoring is essential in production to detect faults or other events in real time and react as fast as possible to fix problems. The data gathered through monitoring can also be used to analyze load trends, understand workloads and compare them with estimations. Thus it is possible to react to load trends before system is overwhelmed.

Logging and monitoring requirements contribute to keep high availability.

The system should include internal monitoring and external monitoring. The former permits immediate identification of a faulty element. The latter controls how system behaves from outside point of view and it usually involves checking availability and response time as it is experienced in the Internet. External monitoring, for instance, could help detect a routing problem that drops packets addressed to the system. This could pass inadvertent for internal monitoring.

3.4 Availability Requirements

Certain measurable requirements will be necessary to eliminate single points of failure and improve availability:

- If a component is replicated, failure of one instance shall not affect other instances.
- Every component shall be stateless and any state that needs to be maintained shall be stored in a data storage mechanism accessible by all replicated components.
- All components shall execute without change when a load balancer is added.
- For all applications, a level of criticality shall be defined. This is going to be useful when defining Service Level Agreements (SLA).
- Every component shall be assigned a level of criticality based on the applications it supports: some components may not be critical and SLA could be less stringent.
- The level of infrastructure availability shall determine the application availability. An application should not reduce availability.
- Application upgrades shall not require service interruption.
- The application shall gracefully behave during component or data center failover handling it transparently for end-users.

It is very important to specify the level of availability the system will provide as there are significant architecture implications. For instance, it is different to develop a system that provides 99.9% availability than a system with 99.99% availability. While the first one can achieve its goal with single points of failures, there are little possibilities to achieve 99.99% availability with those single points of failure, without automatic remediation or with a system that is not designed for failures.

3.5 Non-Measurable Requirements

Non-measurable requirements will help understand the order of magnitude to which business entities and events should be able to scale. These are useful to guide architects and developers when the application

architecture is defined and components are designed. They are also useful to guide quality assurance engineers when designing workloads and preloaded data for performance test.

During analysis, design, and coding there are continuous decisions of trade-off among different qualities: maintainability, flexibility, performance, security, etc. Understanding an entity must scale to millions will push to simpler data models or specific, sometimes difficult to maintain, shortcuts when dealing with these entities. Alternatively, it could lead to smarter decisions of meta-language generation that allows the developers to deal with maintainable scripts that is later transformed into high-performance code. It could also be decided that a NoSQL database is to be used to store entities that will scale to millions.

Regarding availability, specifying a system is 3 nines, or 4 nines, is a non-measurable requirement.

Examples of recommended non-measurable requirements we used:

- The system shall be able to provide 99.9% availability (3 nines).
- The system shall be able to handle hundreds of millions of users (which would be different than developing for thousands of users)

4 Architecture and Design Guidelines

Architecture, design, and coding guidelines help the team keep focus on best known practices that help systems keep good enough performance and scale horizontally with acceptable performance increase.
Developing for parallel processing, stateless components, horizontal scalability even in the data layer, client-centric eventual data consistency [Tanenbaum, 2002], and programming for failures are concepts required for cloud applications.

It is important to understand it is not the same using the cloud as designing for the cloud. In general running a system that was not designed for scalability, or is not programmed for failures, in the cloud, has little, or none, advantage.

The goal is to create an architecture that scales horizontally by adding boxes at all layers achieving linear and infinite scalability as shown in Figure 2.

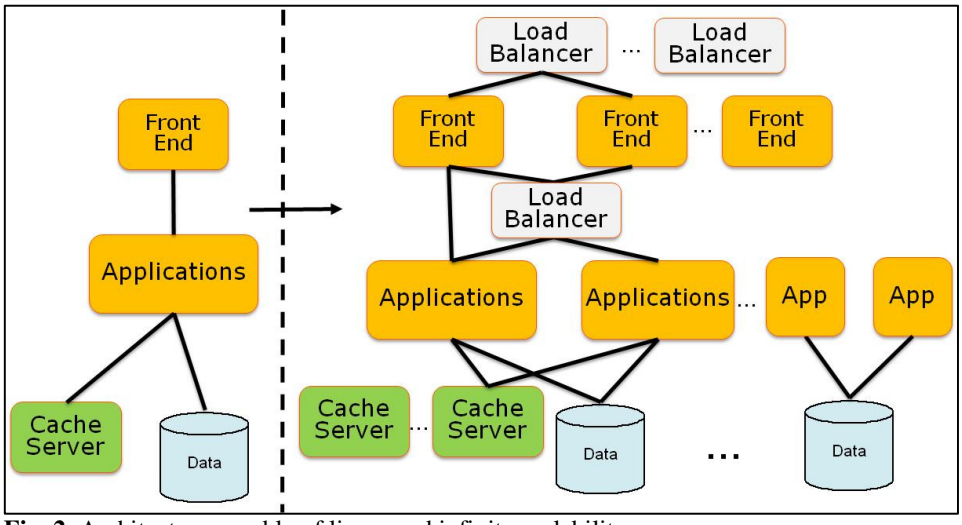

Fig. 2. Architecture capable of linear and infinite scalability.

In scalability reviews, even when a system scales well, it is common to discover eventual consistency problems in write-after-write processes and read-after-write processes [Tanenbaum, 2002]. Figure 3 depicts the problem of read-after-write. This basically occurs because it is not possible to assure that when the read operation arrives, written data is already replicated in all instances. An application should be able to deal with this situation, otherwise, it will not be possible to scale by replicating the database and it will turn into an unavoidable bottleneck.

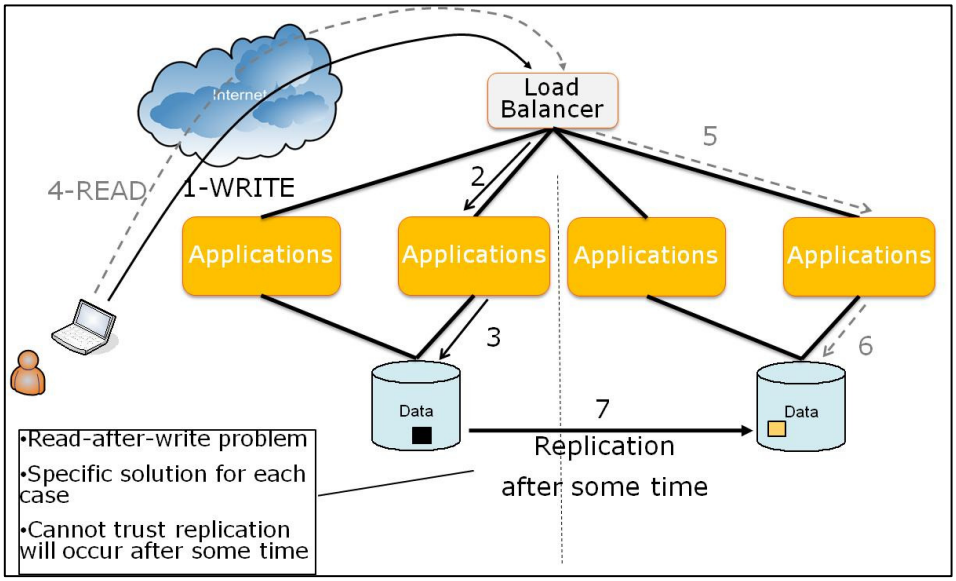

Fig. 3. Eventual consistency problem in read-after-write processes.

The concept of eventual consistency applies to systems with BASE properties: Basically Available, Soft State, Eventual Consistency). Other possible issues in this type of systems are write conflicts. Even though some databases have Anti-Entropy protocols usually conflicts in production require manual intervention if the data model was not carefully designed.

The topic of architecture for scalability is really extensive and encompasses different aspects. In this section it will only be enumerated some typical architectural patterns to consider depending on the application domain (the list is not complete):

- Load Balancer: requests load is balanced by a dispatcher (load balancer) among stateless and replicable workers (or web services in some cases).
- Scatter and Gather: a request is broadcasted by a dispatcher to all workers and results are aggregated or consolidated.
- Result Cache: typical cache solutions usually to avoid access to the database but cache could also store a whole response for a request to avoid further processing.
- Shared Space: a request is stored in a tuple space and workers take a job and the place their response. This is the typical Linda model [Linda in Context, 1989].
- Pipes and Filters: workers are connected through pipes and a request passes through different pipes during its processing until the response is generated.
- Map/Reduce: the algorithm implemented by Google. Hadoop is a widely used open source implementation. This algorithm is mostly used for batch processing but there are some realtime-like implementations [Apache hadoop goes realtime, 2011].
- Bulk Synchronous Parallel: a master coordinates lock-step execution across all workers.
- Execution Orchestrator: schedules ready-to-run tasks among workers.

4.1 Data Sharding

Data Sharding is a technique for horizontal partitioning of data across a number of database nodes that could be in the same or different datacenters. Data Sharding is realized by having a sharding key based on which a shard manager knows which instance to redirect a query.
This is the preferred scalability solution for the database layer both for relational databases (RDBMS) and NoSQL databases.
Sharding can be implemented in the application itself but it can also be made transparent by existing third-party products or by the database technology which already implements it.

4.2 NoSQL Databases

When it is necessary to scale in an infinite manner and the system has to respond fast or deal the huge data, NoSQL databases could be the best option to gain in performance and scalability both of requests and data. NoSQL database that are schema-less are also the preferred option when data can be sparse and can be stored in multiple unpredictable formats.

5 Results

Our product provides common services like live update, provisioning, events, and others to specific applications intended to reach hundreds of millions of users and devices.

This combination of measurable and non-measurable requirements, process and guidelines within the SCAP process helped the team create a scalable product that can keep up with our expected business growth during this and the following years.

After more than a year in production, our system maintained the required level of availability and there was instant notice of any minor incident like detection of response time decrease by the external monitoring system at certain moments. However, we are still working to make availability even better.

Regarding QA, four types of performance tests were designed:

- Performance tests
- Stress tests
- Scalability tests
- Longevity tests

Based on workload analysis and estimations, QA team was able to create representative workloads to test the application. It is also possible to continuously improve the testing workloads based on information obtained from production and discussions with the development team.

Probably the most important achievement is that performance, scalability, and availability are not a scary topic anymore. There is complete awareness in the teams about these requirements and they focus on them from the beginning. As an example, whereas some time ago we used to find out eventual consistency problems after the system was developed and we tried to scale out, today we find these types of problems during architecture reviews before coding begins.

5.1 Performance Test

As mentioned before, four types of performance tests are executed. Performance tests help determine whether the system can achieve its performance targets with the existing hardware. Stress tests are intended to find the system break points and discover weird errors like memory leaks, race conditions, deadlocks, etc. Scalability tests are intended to understand application scalability profile and also to find the best ratios among servers in different layers. By determining the optimum ratios, it is possible to define a scale unit for the system. Longevity tests are meant to understand system behavior running at average load during long periods of time.

All of these tests are necessary for services and the success of these tests implies there are more possibilities to achieve good enough service availability even under load peaks and there is less risk of violating established SLA's.

The results of stress and performance tests refer both to software and hardware capacity. This means that usually performance can be increased by software modifications or by adding servers.

Scalability tests demonstrate how, by adding servers (horizontal scalability), capacity and throughput are increased. Its results usually depend on the software and system configuration. If a bottleneck is found in the DB, it can be improved by scaling up (adding hardware to the DB server). However, systems should allow DB horizontal scalability, by replication or partitioning. This may require application re-architecture.

Based on the workload analysis and workload estimations, QA team was able to design representative workloads to test the system. Then, for each version, we are able to have a good idea of how our system would behave under those workloads and determine whether our system can achieve performance targets or not. Additionally, we can measure how new features affect performance baselines. Results usually open the way to discussions within the team that allow managers taking release decisions with richer data. Additionally, test results are useful to identify further opportunities for improvement and allow management and technical leaders to take better decisions about where to invest effort to draw the best results.

5.2 Performance and Scalability Results

Performance tests allowed us to keep control of system throughput and response time all the time. This helps perform more conscious load monitoring against thresholds.

Figure 4 shows different measurements of one of our services during more than one year. Measurements and dates have been normalized in order to hide real values. This figure shows how, through different development sprints, throughput, expressed in Requests per Second (RPS), changes and QA team provides enough information to detect problems and improve performance so that system is not affected in production. Additionally, the figure shows how system throughput increased during this period.

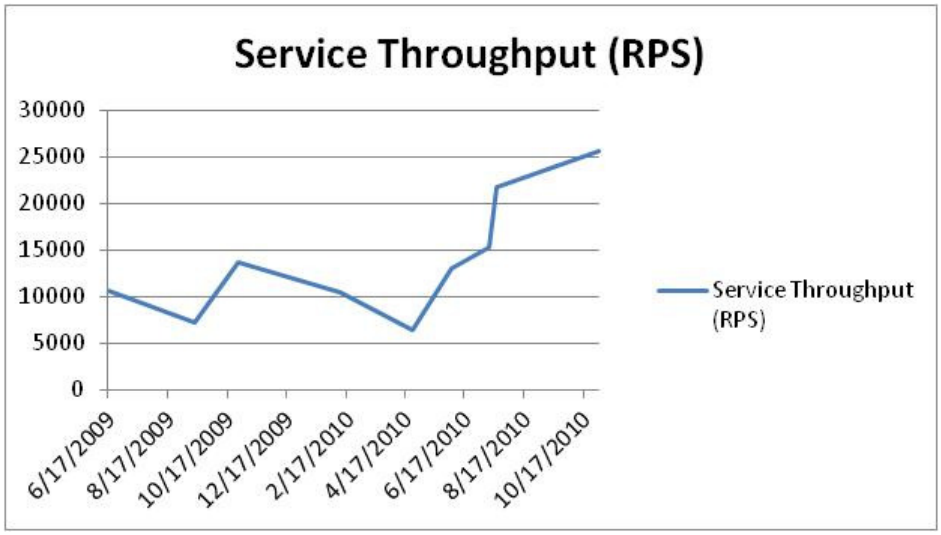

Fig. 4. Continuous throughput measurement in requests per second (RPS) in one year.

It is important to note that all performance tests are executed on a baseline capacity in order to produce comparable results.

In other operation not shown here, after detecting a performance issue in one performance test cycle, the development team was able to fix the problem and deliver 8x throughput improvement while response time dramatically fell. This implied a significant reduction in number of servers required to run the system providing the necessary capacity. At the end, this means lower total cost of ownership.

In different scalability tests our product demonstrated a great level of scalability. We defined a scale unit composed by a set of N virtual servers delivering certain throughput. As we tested, performance increased almost linearly per scale unit added. We needed to execute a special test borrowing a huge number of servers to reach our next scalability inflection point. Figure 5 shows the results of scalability tests executed beyond a baseline called 1x, with N servers. It was not until 10x that we were able to find our inflection point where the scalability curve begins to clearly separate from linear scalability. For confidentiality reasons, numbers have been normalized. After we found and solve the bottleneck, our product was able to have even better scalability close to the ideal “infinite” scalability for the load we expect in the following years. This scalability level can even occur across different datacenters.

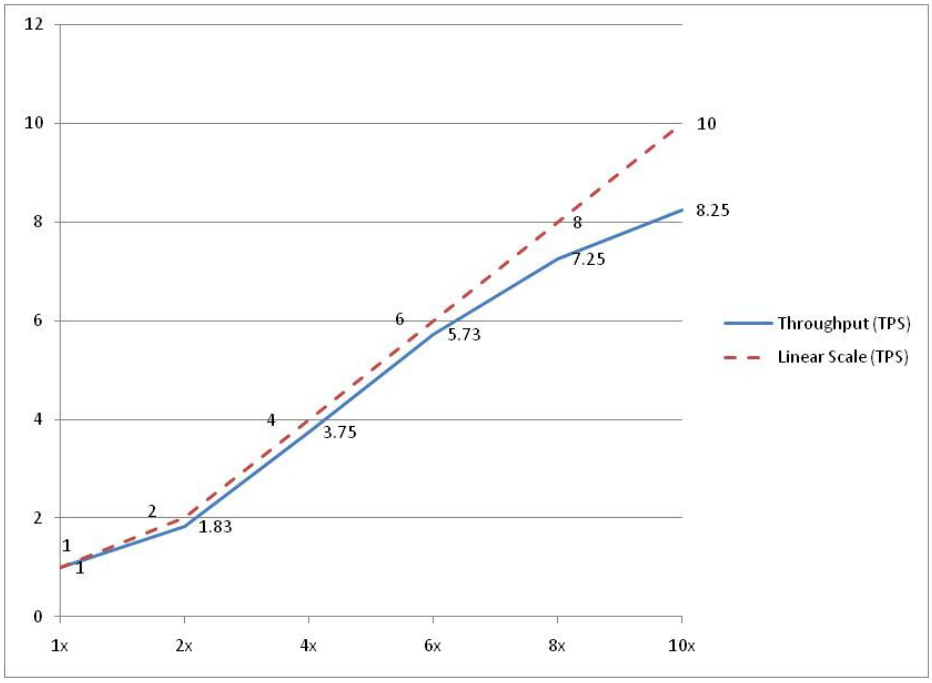

Fig. 5. System scalability results beyond a selected baseline of 1x.

All tests were executed using database replication in master-master mode.

In a further test the team probed this service can run in two different datacenters in parallel, in active-active mode without affecting data consistency. Throughput kept invariable and it only decreased by 10% when we injected latency corresponding to the maximum geographical distance.

There are other services where scalability with database replication was not straightforward as the design suffered from eventual consistency problems. A re-design was required to make it scale with master-master database replication. As a general rule, software architects should review all cases of write-after-write and read-after-write in order to identify possible eventual consistency problems in their designs.

Since we began applying SCAP process, our application performance in a minimum scale unit increased more than 20x in two years. We were able to control our performance and continuously improve it. Having more performance per scale unit helped us reduce capital costs.

Today our application can scale out to deal with our expected loads during the following years.

5.3 Availability Results

As mentioned above, we measured the required level of availability during more than one year. This system quality not only depends on software but also on infrastructure. However, software should not affect negatively the availability provided by the infrastructure.

At this moment we are working to improve it by increasing automation and improving self-remediation capabilities.

5.4 Discussions on Performance Targets and Tests Results

SCAP process improved the quality of team discussions regarding system performance, availability, and scalability by providing data and focusing the discussions.

First of all, the process makes it possible to reach consensus on what performance targets are for each application and under specific workloads. Different stakeholders can focus their discussion on how they think a business entity will generate a business event. Once, they agreed on that, performance target calculation is straightforward with a well known and accepted formula. Whenever a target is not achieved, a meeting is called to discuss about the test, the results, and how they may affect the business while that software version is in production. Then decision is made about approving or not a release. Moreover, at that point there are no discussions about validity of performance targets.

It is also possible to understand the effect of changes between versions. It is possible to detect low performance trends and take actions for next releases.

Performance test results help managers and technical leaders identify bottlenecks and evaluate whether to invest effort in solving them or not.

Finally, it is possible to provide IT with concrete data when it must be decided about the hardware to invest in production. This situation permits to save costs on spare capacity that may be installed and not utilized while at the same time minimizing risks of not having enough capacity to deal with peaks and bursts. In the cloud it is essential that a system can horizontally scale in order to take advantage of cloud computing benefits. Our tests help determine the scale unit that sets the basis to scale.

5.5 Consequences in the Cloud

Having data about workload analysis, system capacity, scalability profile, and real workload in production was important to define a fix capacity base for a system and estimate future needs for burst capacity. This was useful to determine future budget required for operations and take decisions whether the revenue will pay off.

A company could decide to have a hybrid cloud infrastructure made of a base infrastructure hosted locally for normal execution and an Infrastructure-as-a-Service provider for bursting capacity. This is usually the model that optimizes costs. Performance tests and capacity planning allow determining the needed base infrastructure and the maximum load supported. Scalability tests are necessary to understand whether the system can scale by adding new servers.

Having the scalability profile was necessary to obtain the optimum scale unit for the application under certain workloads.

5.6 Integration with Agile Methodologies

SCAP process perfectly fits in agile methodologies and permits a continuous performance control across different sprints.

It defines different milestones:

- SCAP0: Scalability Assessment. This is located at the end of the main requirements gathering phase that is input for architecture definition. It includes:
    - Special requirements.
    - Identification of scalable entities and events.

    - Etc.
- SCAP1: Architecture Readiness. Right before beginning architecture. The following tasks are required:
    - Workload analysis ready.
    - Requirements ready.
    - Business performance targets established.
- SCAP2: Architecture Review. It includes:
    - Scalability and performance review.
    - Implementation performance targets.
    - Total cost of ownership analysis.
- SCAP3: Design readiness check and design review. This milestone occurs for every sprint before beginning features design and right after design is done. It includes:
    - Updated workload analysis.
    - Updated implementation performance targets.
    - Scalability and performance review.
    - Total cost of ownership analysis review. This may change after implementing a new feature.
- Performance tests are executed for every sprint after features implementation. This is part of SCAP3.
- SCAP4: Beta / Gold Release approval. This milestone is an input for the release approval criteria. It requires the following input:
    - Performance test results
    - Capacity Plan
    - Monitoring Plan
    - Business information plan
- SCAP5: Production services operations. This is a continuous activity while system is executing in production. It includes:
    - Monitoring
    - Workload feedback

SCAP3 milestones must be repeated for every development sprint.

5.7 Opportunities for Improvement

After running this process for more than one year we found many opportunities for improvement. The most important was the cost of executing these tests. We have progressed in performance test automation and our final goal is to automatically execute nightly performance tests being able to early detect performance issues thus giving the development team more time to solve them.

Regarding workload analysis, it might be argued that there is a high level of guessing when estimating load. However, this is also true when estimating development effort and this has proved been a great practice. Additionally, as we have more information from production, workload estimations become more accurate.

We have improved our availability with self-remediation capabilities and we are still working on them.

6 Conclusions

This paper proposes a holistic approach to manage specific system qualities: performance, scalability, and availability. A process, SCAP, is proposed to combine measurable requirements, non-measurable requirements, activities and assets, and architecture guidelines to manage these system qualities. This process

is not an alternative for performance engineering practices and methodologies to handle non-functional requirements. It is a higher level approach that can contain them.

Disaster recovery and geographical distribution with multiple availability zones were not discussed in this paper in order to reduce the scope but achieving the mentioned requirements is a pre-condition for these capabilities.

Here, it is preferred to discuss about managing performance, scalability, and availability instead of just predicting or evaluating it. “Managing” is a more comprehensive term that encompasses prediction, generation of input to the product architecture and design, generation of input for capacity planning, evaluation, reporting, and monitoring.

It may be argued that results are not a direct consequence of this process but just what skillful developers did. However, the continuous focus on scalability, performance, and availability, the discussions around workload estimations and issues found during performance tests, the awareness of possible problems after workload estimations, have created an environment where there is continuous improvement of these quality attributes. Additionally, identified bottlenecks and errors helped the team invest the effort in solutions that would yield highest improvement with the corresponding total cost of ownership reduction in infrastructure.

As of today, this process proved to be useful to avoid problems in production even when eventual bursts occurred. As it was explained, our system demonstrated to be highly scalable and a great level of availability.

References

Chung L., Sampaio do Prado Leite J. C., On Non-Functional Requirements in Software Engineering, Book Chapter, p.363-379, Conceptual Modeling: Foundations and Applications, Springer Berlin / Heidelberg, (2009)

Amazon Web Services, online: http://aws.amazon.com (2012)

Rackspace, online: http://www.rackspace.com (2012)

Google App Engine, online: https://appengine.google.com (2012)

Windows Azure, online: http://www.windowsazure.com (2012)

Compuware, When More Web Site Visitors Hurt Your Business: Are You Ready for Peak Traffic?, Dec. (2009), online: http://www.gomez.com/wp-content/downloads/when-more-website-visitors-hurt-your-business-are-you-ready-for-peak-traffic.pdf

Mylopoulos J., Chung L., and Nixon B., Representing and using nonfunctional requirements: a process-oriented approach, Software Engineering, IEEE Transactions on , vol.18, no.6, pp.483-497, Jun (1992)

Bennett A. J. and Field A. J., Performance Engineering with the UML Profile for Schedulability, Performance and Time: A Case Study. In Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '04). IEEE Computer Society, Washington, DC, USA, 67-75. (2004)

Mell P. and Grance T., The NIST Definition of Cloud Computing – Recommendations of the National Institute of Standards and Technology, National Institute of Standards and Technology, US. Department of Commerce, Special Publication 800-145, September (2011), Online: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

OpenStack, online: http://www.openstack.org (2012)

Michael M., Moreira, J.E., Shiloach D. and Wisniewski R.W., Scale-up x Scale-out: A Case Study using Nutch/Lucene, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, vol., no., pp.1-8, 26-30 March (2007)

CIM, Common Information Model, December (2011), online: http://www.dmtf.org/standards/cim

ITU-T, Information Technology - Open Systems Interconnection - System Management: Alarm Reporting Function, ITU-T X.733, (1992)

Tanenbaum A. and van Steen M., Distributed Systems, Principles and Paradigms, Prentice Hall, (2002)

Gelernter D. and Carriero N., Linda in Context, Communications of the ACM, v.32 n.4, p.444-458, April 1989

Borthakur D. et al., Apache hadoop goes realtime at Facebook. In *Proceedings of the 2011 international conference on Management of data* (SIGMOD '11). ACM, New York, NY, USA, 1071-1080. DOI=10.1145/1989323.1989438 http://doi.acm.org/10.1145/1989323.1989438 (2011)