Impact of *pay-as-you-go* cloud platforms on software pricing and development: a review and case study

Fernando Pires Barbosa and Andrea Schwertner Charão

Programa de Pós-Graduação em Informática,
Universidade Federal de Santa Maria, RS - Brasil
`fernando.pires.barbosa@gmail.com`,
`andrea@inf.ufsm.br`

`http://www.ufsm.br/ppgi`

Abstract. One of the major highlights of cloud computing concerns the *pay-as-you-go* pricing model, where one pays according to the amount of resources consumed. Some cloud platforms already offer the *pay-as-you-go* model and this creates a new scenario in which the rational computing resource consumption gains in importance. In this paper, we address the impact of this new approach in software pricing and software development. Our hypothesis is that hardware consumption may impact directly on the software vendor profit and thus it can be necessary to adapt some software development practices. In this direction, we discuss the need to revise well-established models such as COCOMO II and some aspects related to requirements engineering and benchmarking tools. We also present a case study pointing that disregarding the rational consumption of resources can generate wastes that may impact on the software vendor profit.

Keywords: Cloud Computing, Software Pricing, Cloud Platform, Software Engineering

1 Introduction

Cloud computing is a new computing paradigm based on economies of scale, which predicts the existence of a dynamically scalable infrastructure, where resources are allocated and delivered on demand over the Internet [6]. One of cloud computing highlights is its pricing model, also known as *pay-as-you-go*. In this model, IT resources are offered in an unlimited way and one pays an amount according the actual resources used for a certain period (similarly to the energy pricing model) [2].

The cloud vendors offer different kind of services, including *IaaS*, *DaaS*, *PaaS* and *SaaS* [9]. Although one can deploy applications on any of the layers, the more natural option for software developers is the platform as a service (*PaaS*) [2]. Some *PaaS* vendors currently adopt the *pay-as-you-go* model and, in some cases, one can clearly see the financial impact of software optimization techniques applied to resource consumption. In AppEngine platform, for example, studies indicate that using cache strategy instead of direct database access can reduce by about 20 times the total cost of the operation [2]. Issues like that create a new scenario for software developers, who may have to adapt some of the practices they currently use.

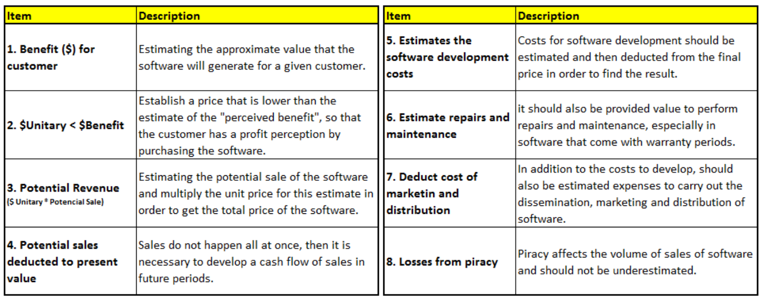
In this new scenario, rational resource consumption become a strategy to reduce the amount of hardware used by applications and therefore reduce the amount to be paid to the vendor's platform. In this paper, we review software pricing issues facing this scenario and analyze some of the practices used in software development, indicating whether and how they are affected by this new reality. The paper is organized as follows: Section 2 focus on software pricing while Section 3 analyze software development aspects. The analysis is focused on: i) software development estimates (with COCOMO II), ii) requirements engineering (ISO/IEC 25010) and iii) benchmarking tools (SPEC). Section 4 presents a case study aiming to identify whether a system developed without concern for the rational resource consumption can generate resource wasting that may result in financial loss if the system is distributed through a *pay-as-you-go* cloud platform. Section 5 presents our final remarks.

2 Software Pricing

Software pricing has been discussed for several years in many ways. Since the '90s there has been studies on establishing a fair price for software [5] and new issues have been addressed recently. One of the recent research topics concerns the differences between the traditional, perpetual license model and the new software as a service (SaaS) model [12]. Beyond the SaaS pricing model issues, different studies on software pricing have been developed, including: models using the value added to client's business [12], studies based on stock market [16], pricing based on cost accounting [19] and use of price sensitivity in order to identify features that should be prioritized [10]. Even with so many different studies, software pricing involves basic elements that can be applied to all models [5], as summarized in Table 1).

Considering the aspects in Table 1, the first item affected by *pay-as-you-go* cloud platforms is number *3. Revenue Potential.* In the traditional model, the hardware required to run the software is a customer obligation. In the cloud model, it will be probably an obligation applied to software vendor. If the software is based on a *pay-as-you-go* platform, the software vendor company will have part of the revenue gains spent to pay the platform vendor.

Establishing a final software price in this scenario involves estimating not just the software revenue potential, but also the hardware resources required to use it. That changes the way we face the hardware resources throughout the software development process: in the *pay-as-you-go* model, one must design the software to use minimal hardware resources, since it will directly impact your profit. This affects another item in Table 1: number *5. Estimate software*



Item	Description	Item	Description
1. Benefit ($) for customer	Estimating the approximate value that the software will generate for a given customer.	**5. Estimates the software development costs**	Costs for software development should be estimated and then deducted from the final price in order to find the result.
2. $Unitary < $Benefit	Establish a price that is lower than the estimate of the "perceived benefit", so that the customer has a profit perception by purchasing the software.	**6. Estimate repairs and maintenance**	it should also be provided value to perform repairs and maintenance, especially in software that come with warranty periods.
3. Potential Revenue ($ Unitary * Potencial Sale)	Estimating the potential sale of the software and multiply the unit price for this estimate in order to get the total price of the software.	**7. Deduct cost of marketin and distribution**	In addition to the costs to develop, should also be estimated expenses to carry out the dissemination, marketing and distribution of software.
4. Potential sales deducted to present value	Sales do not happen all at once, then it is necessary to develop a cash flow of sales in future periods.	**8. Losses from piracy**	Piracy affects the volume of sales of software and should not be underestimated.

Table 1. Software pricing issues [5].

development cost. Section 3 presents an analysis on software development and the first aspect analyzed, in Section 3.1, is how one of the most used software cost estimation models can face this situation.

3 Impact on software development

Due to changes in software pricing addressed in Section 2, we review some aspects of software development that could also change face to the *pay-as-ou-go* model. In this section, we analyze software cost estimation (with COCOMO II), requirements engineering (with ISO/IEC 25010 and the traditional approach of non-functional requirements based on quality standards) and, at last, benchmarking tools (the traditional SPEC benchmark suites and new research works that are going on).

3.1 Software cost estimation (COCOMO)

The first studies on software cost estimation begun in '60s and there has been significant progress since then. Several models have been proposed during the '70s and '80s and some of them have been gradually improved and adapted until today. One of the most used models is COCOMO II (Constructive Cost Model), which is the latest major extension to the original COCOMO (COCOMO 81) model published in 1981 and has several extensions as shown in Fig. 1.

COCOMO II estimates the effort using a person-month value based on 22 items split into 5 software scale drivers and 17 software cost drivers as shown in Table 2. For each item is assigned a value and, the higher this value, the greater the effort required. *Required Software Reliability* (RELY), for example, means the extent to which the software must perform its intended function over a period of time. It ranges from *very low* to *very high*. If the effect of a software failure is only a slight inconvenience, the RELY value is *very low*. If a failure would risk human life then RELY is *very high*.

The *platform* items refers to the target-machine hardware and infrastructure software. *Execution Time Constraint* (TIME) is expressed in terms of the percentage of available execution time to be used by the software. *Main Storage*

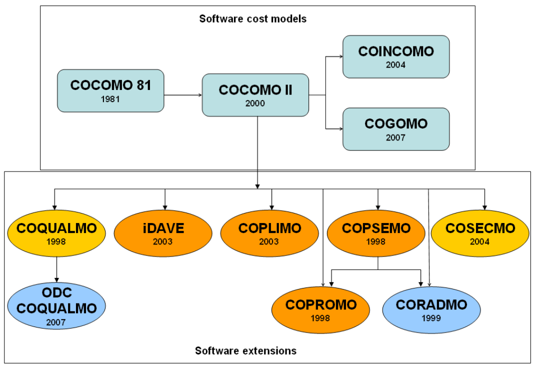
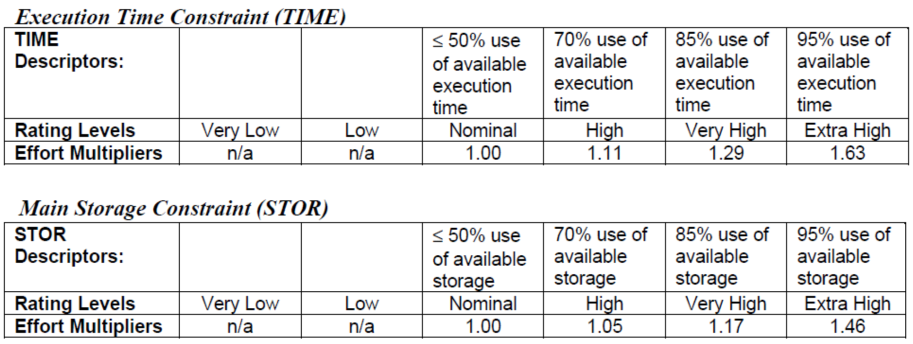


Fig. 1. COCOMO extensions, adapted from [3]. Dates mean the first paper published.

Constraint (STOR) represents the degree of main storage constraint imposed on the software. Fig. 2 shows the rating ranges for TIME and STOR.


Execution Time Constraint (TIME)

TIME Descriptors:			≤ 50% use of available execution time	70% use of available execution time	85% use of available execution time	95% use of available execution time
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	n/a	n/a	1.00	1.11	1.29	1.63

Main Storage Constraint (STOR)

STOR Descriptors:			≤ 50% use of available storage	70% use of available storage	85% use of available storage	95% use of available storage
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	n/a	n/a	1.00	1.05	1.17	1.46

Fig. 2. TIME and STORE rating ranges (COCOMO Model Definition Manual [4]).

In a *pay-as-you-go* platform, it would be impossible to use the approach based on percentage of available resources as shown in Fig. 2. In a cloud environment, by definition, there is no maximum available resource amount. Resources are provided on demand according to user needs. So it does not make sense to range TIME and STOR based on an available resource percentage.

In a paper on the impact of the cloud model in software engineering [8], the authors propose a change in COCOMO 81, suggesting the creation of a new software class called *Cloud Computing*, to represent the complexity added by cloud platforms. The point is that this steady increase in software complexity is one of the reasons that led replacing COCOMO 81 by COCOMO II. That is, the problem that the article is meant to solve has been solved yet (by COCOMO II). However, even COCOMO II need to be revised. Not just about the cloud

	Driver Kind	Driver item	Range					
			Very Low	Low	Nominal	High	Very High	Extra High
Software Cost Drivers (Linear Influence)	Product	Required software reliability	x	x	x	x	x	
		Database size		x	x	x	x	
		Product Complexity	x	x	x	x	x	x
		Developed for Reusability		x	x	x	x	x
		Documentation match to lifecycle needs	x	x	x	x	x	
	Plataform	Time Constraint			x	x	x	x
		Storage Constraint			x	x	x	x
		Platform Volatility		x	x	x	x	
	Personnel	Analyst capability	x	x	x	x	x	
		Programmer Capability	x	x	x	x	x	
		Personnel Continuity	x	x	x	x	x	
		Application Experience	x	x	x	x	x	
		Platform Experience	x	x	x	x	x	
		Language and Toolset Experience	x	x	x	x	x	
	Project	Use of Software Tools	x	x	x	x	x	
		Multisite Development	x	x	x	x	x	x
		Required Development Schedule	x	x	x	x	x	x
Software Scale Drivers (Exponential Influence)		Precedentedness	x	x	x	x	x	x
		Development Flexibility	x	x	x	x	x	x
		Architecture / Risk Resolution	x	x	x	x	x	x
		Team Cohesion	x	x	x	x	x	x
		Process Maturity	x	x	x	x	x	x

Table 2. COCOMO cost and scale drivers. Highlight on *platform* factor.

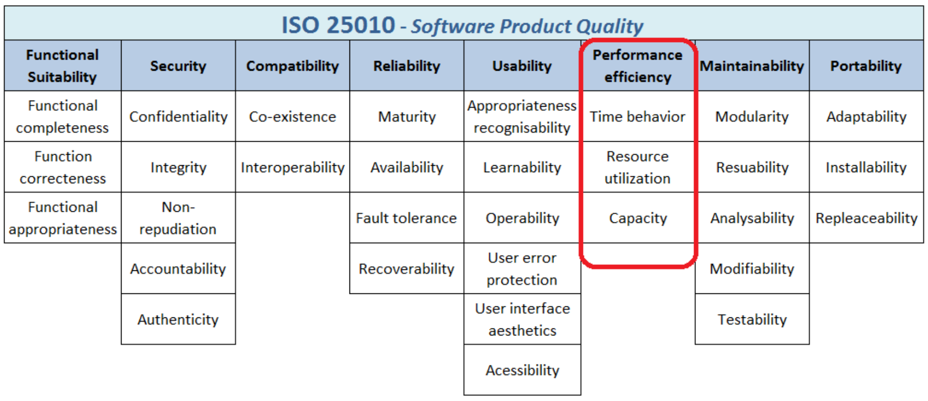
environment and its complexity, but mainly about the *pay-as-you-go* model. In such model, the more resources you consume the less its profit. So there is no way to establish a value for the items TIME and STOR based on a percentage of available resources. They must be revised to take into account the rational consumption of resources, even though they are always available. Another question to be addressed on this aspect is: the tools and models currently used allow you know beforehand the amount of resources that will be consumed by a particular software? And how do you know if a particular piece of code should or should not be optimized on this perspective? Section 3.2 discusses these issues.

3.2 Requirements and software engineering (ISO/IEC 25010)

Requirements engineering is generally accepted to be the most critical and complex process within the software development process [15]. It makes sense, since the requirements are used to describe software features and its behavior. Requirements engineering is mostly performed in the beginning of the software development cycle, but its activities cover the entire cycle in order to detail the software features [15]. Requirements are commonly classified as *functional* and *non-functional*. A functional requirement specifies an action performed by a system without considering its environment. Non-functional requirements describes just the characteristics such as environment, platform, performance, constraints, reliability, etc.

Functional and non-functional requirements are put together in a Software Requirement Specification (SRS) document. It contains details on each feature and information about how the application must interact with the system environment, considering issues such as response time, availability etc. SRS document is the one which will guide the entire software development process.

Being an important issue, requirements are also taken into account by software quality area. ISO/IEC 25010 replaced ISO/IEC 9126 in software quality standards. It defines 8 characteristics for product quality. There are also 31 sub-characteristics as shown in Fig. 3 [11].



ISO 25010 - *Software Product Quality*

Functional Suitability	Security	Compatibility	Reliability	Usability	Performance efficiency	Maintainability	Portability
Functional completeness	Confidentiality	Co-existence	Maturity	Appropriateness recognisability	Time behavior	Modularity	Adaptability
Function correcteness	Integrity	Interoperability	Availability	Learnability	Resource utilization	Resuability	Installability
Functional appropriateness	Non-repudiation		Fault tolerance	Operability	Capacity	Analysability	Repleaceability
	Accountability		Recoverability	User error protection		Modifiability	
	Authenticity			User interface aesthetics		Testability	
				Acessibility			

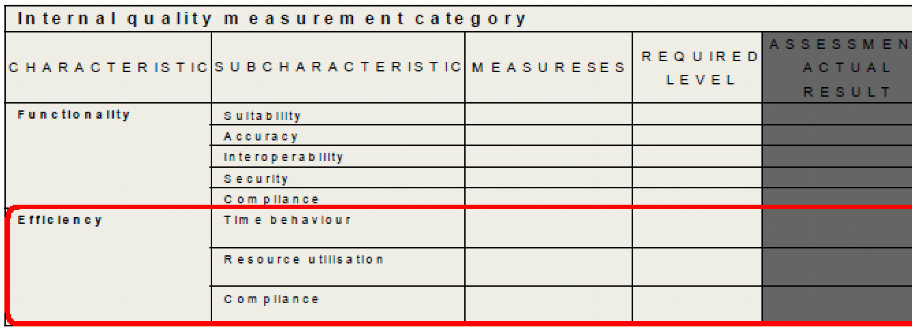
Fig. 3. ISO25010 characteristics. Highlight to *efficiency* characteristics.

Performance efficiency characteristic is defined by ISO 25010 as "the *performance relative to the amount of resources used under stated condition*". That is the quality attribute related to resource consumption. It has 3 sub-characteristics: *time behavior*, related do response time; *resource utilization*, amount and type of resources used; *capacity*, maximum limits of the product (such as concurrent users, size of database, throughput transactions etc.)

Thus, in software engineering, performance and resource consumption requirements are treated as non-functional requirements and have a direct relation with quality attributes, specifically the *Performance Efficiency* characteristic. The *efficiency* measurement is made by objective criteria, which are established in advance and must be attended by the software. An example criteria is: *"on a server with 1GB RAM and two 1,8GHz processors operating with a load of 400 concurrent users, 95% of all requests must return within 5 seconds and 90% of them in up to 3 seconds using up to 50% of CPU and memory available"*.

After detailing the *efficiency* criteria in non-functional requirements at SRS, it is possible to lead measurements to check if software meets the criteria or not. It can be done using forms like the one at Fig. 4. This is how the resource consumption issues are treated in traditional software development process.

Traditional approach induces the software requirement process to identify features with huge concurrent use. That is because one must inform the software designers which features they should provide a special treatment in order to ensure the response time required by quality criteria. In the *pay-as-you-go*



Internal quality measurement category				
CHARACTERISTIC	SUBCHARACTERISTIC	MEASURESES	REQUIRED LEVEL	ASSESSMEN ACTUAL RESULT
Functionality	Suitability			
	Accuracy			
	Interoperability			
	Security			
	Compliance			
Efficiency	Time behaviour			
	Resource utilisation			
	Compliance			

Fig. 4. Example of form used to software *efficiency* assessment [17].

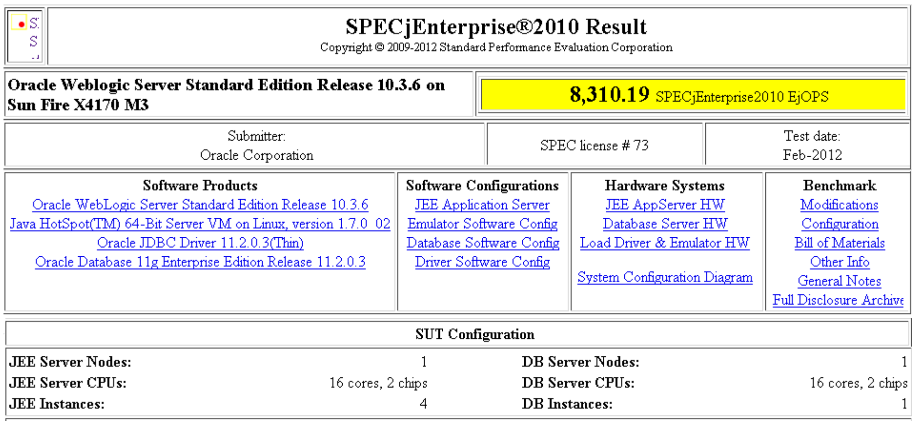
model, the amount of resources consumed affects software vendor profit. Therewith, resource consumption is no longer just a matter of quality and must be viewed in a more strategic way. That brings a new issue to software requirements engineering: estimating the amount of resources that will be consumed by the software. Optimize an application to consume the lower amount of resource is different than optimize it to attempt quality criteria such as response time or throughput. The new question to be addressed in the software requirement elicitation process is: *"which features will be used more often?"*.

The answer to these questions indicates which features should be optimized as an strategy to reduce the amount paid to platform vendor. That brings a new concern to software developers about the optimization: *optimizing features with biggest potential to consume resources*. All of that indicates another software development aspect to be reviwed: the performance benchmark tools.

3.3 Benchmarking tools (SPEC)

Performance evaluation of computer systems has been studied for several years and one of the most appreciated issues are the benchmarking tools. Until the '80s, the main measuring instruments were MIPS and Mflops (both related to CPU speed). But then the systems complexity required new tools [7] and that has led to corporations like SPEC [18]. SPEC was formed to establish, maintain and endorse a standardized set of relevant benchmarks, developing and regulating benchmark suites. SPEC has more than 60 members and also reviews and publishes submitted results from them. Among benchmark suites provided by SPEC, Java benchmarks are the ones with closer relationship to traditional software development. The *SPECJEnterprise2010* benchmark is the most comprehensive of them, since it considers the whole J2EE specification (including Web Server, Application Server, Database Server and JMS System).

SPECJEnterprise2010 is the third generation of the SPEC organization's J2EE industry standard benchmark application. Its performance metric is EjOPS (Enterprise jAppServer Operations Per Second). Earlier generations also used a price/performance metric but it was removed and now one must calculate price/performance separately. That can be done using EjOPS and the BOM (Bill of Materials) used to reproduce the results. Fig. 5 shows how the results are provided by SPEC.



SPECjEnterprise®2010 Result

Copyright © 2009-2012 Standard Performance Evaluation Corporation

Oracle Weblogic Server Standard Edition Release 10.3.6 on Sun Fire X4170 M3

8,310.19 SPECjEnterprise2010 EjOPS

Submitter: Oracle Corporation	SPEC license # 73	Test date: Feb-2012

Software Products	Software Configurations	Hardware Systems	Benchmark
Oracle WebLogic Server Standard Edition Release 10.3.6	JEE Application Server	JEE AppServer HW	Modifications
Java HotSpot(TM) 64-Bit Server VM on Linux, version 1.7.0_02	Emulator Software Config	Database Server HW	Configuration
Oracle JDBC Driver 11.2.0.3(Thin)	Database Software Config	Load Driver & Emulator HW	Bill of Materials
Oracle Database 11g Enterprise Edition Release 11.2.0.3	Driver Software Config		Other Info
		System Configuration Diagram	General Notes
			Full Disclosure Archive

SUT Configuration

JEE Server Nodes:	1	**DB Server Nodes:**	1
JEE Server CPUs:	16 cores, 2 chips	**DB Server CPUs:**	16 cores, 2 chips
JEE Instances:	4	**DB Instances:**	1

Fig. 5. Example of *SPECJEnterprise2010* result provided by SPEC.

Traditionally, hardware resources are a customer's obligation. In that case, EjOPS metric can help software vendors to suggest hardware specification to be acquired. But if one deploys software through a cloud model, the hardware will probably be in an abstraction layer which is not visible at the customer point of view. Furthermore, the amount spent on these resources will likely to be paid by the software vendor. In this situation, the EjOPS metric itself makes no sense if there is no related price. The parameters required to evaluate a *pay-as-you-go* cloud platform should be different than ones currently used by SPEC. As shown in Fig. 5, SPEC results include hardware and software specification. That is not relevant in a cloud platform evaluation, which should present something like "how much it will be spent to run a specific workload". There is also another perspective to be considered: benchmark metrics are generally related to processing power and, in *pay-as-you-go* cloud model, there are other billable items such as total storage used, input and output data transfer. Those items should also be covered by a cloud platform benchmark.

CloudCMP is one of the first efforts in developing a new cloud benchmark suite [13]. The first *CloudCmp* results were published in 2010 at the Conference on Internet Measurement in Australia and the benchmark was publicly released in November, 2011 [14]. CloudCmp published data covers some of major cloud vendor such as *Amazon*, *Microsoft*, *Google* and *Rackspace*. Even not addressing only platforms [1], as expected for such research on cloud evaluation, the metrics

[1] Only *Google AppEngine* is really a cloud platform (*PaaS*). The other ones are most like *IaaS*.

initially proposed by *CloudCmp* adhere to the new scenario we describe in the present paper. *CloudCmp* compares items like *benchmark finishing time* vs. *cost* or *scaling latency* vs. *cost*, which are closely connected to what would be necessary to choose a cloud platform vendor. But, as the paper itself warns, there is still a lot of work to do. Some of the future work is to build performance prediction models based on CloudCmp's results to enable cloud provider selection for arbitrary apps. That could be used at the beginning of the software development process as a strategy to estimate the amount to be paid to platform vendor.

4 Case study: SIE ERP system

To study how the changes highlighted in Section 3 could interfere in the outcome of the software development process, we carried out a case study using an ERP system named SIE. This system is targeted to academic institutions and is currently deployed in more than 20 Brazilian universities. SIE development started in mid 1999, using Borland Delphi and a multi-tier architecture where the business rules are processed via RPC calls in one or more centralized application servers [1]. The whole ERP has around 2.500 tables accessed by more than 4.000 applications that work seamlessly to manage different business functions such as academic and student management, contract and inventory management, human resources, finance/accounting, etc.

The specific purpose of our study is to check whether and how a software developed without concern on rational resource consumption can generate wastes that, in a *pay-as-you-go* cloud platform, will directly impact the software vendor profit.

4.1 Case study setup

The ideal scenario to perform measurements over a *pay-as-you-go* platform would require that SIE was hosted on a cloud platform. This scenario, however, is not feasible within the scope of this work, because the actual system would need to be developed with the architecture and/or programming language of the platform.

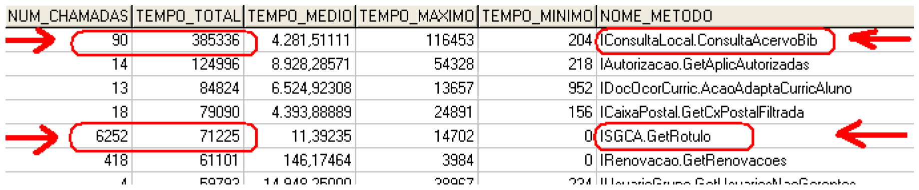
As that scenario is unfeasible, we chose to monitor actual usage of SIE in one institution (Universidade Federal de Santa Maria – UFSM), using its production environment. This approach allows us to collect fairly comprehensive information, since the SIE ERP is widely used in UFSM.

The measurement of effectively consumed resources should track all the billable items of a *pay-as-you-go* platform. Monitoring at this level of detail would not be feasible within the scope of this work, so we chose to monitor the *response time* of each system feature. Then we consider this response time as an indication of resource consumption as follows: the longer response time of a feature, the greater the amount of resources it consumed.

In an ERP like SIE, the response time can be affected by issues such as: CPU usage on the application server; the amount of data transferred between the server and client side; the system's CPU and disk usage on the database server; and the volume of information stored at database along with the required indexes. Although these aspects may keep some similarity with *pay-as-you-go* billed items, measuring the response time is not a substitute for complete detailed resource monitoring. However, for the purpose of this study, the *response time* can be used without major losses.

4.2 Measurement and analysis

To perform the measurements and collect data, we modified SIE's source code to log any RPC call made to the server layer. The log contains, among other information, the name of the RPC method called and the time each call took to be processed. The change was applied to the UFSM's production environment [2] and kept alive for a 20 minutes period. This time was enough to collect data on more than 35.000 RPC calls, made by a total of 58 users, who used 62 different system applications, resulting in calls to 602 different RPC methods. With that log data, it was possible to obtain information as shown in Fig. 6, which lists some of the SIE RPC methods and its response time. Taking a look at that list, one observes that the method named *IConsultaLocal.ConsultaAcervoBib* has a considerable total response time (385.336 *msec*) and it was called 90 times during the monitoring period. Similarly, the method named *ISGCA.GetRotulo* got 6.252 calls and a total response time of 71.225 *msec*.



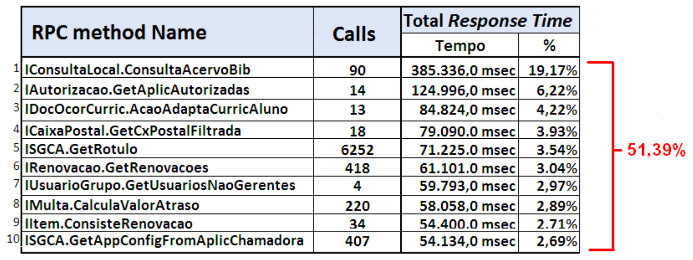
NUM_CHAMADAS	TEMPO_TOTAL	TEMPO_MEDIO	TEMPO_MAXIMO	TEMPO_MINIMO	NOME_METODO
90	385336	4.281,51111	116453	204	IConsultaLocal.ConsultaAcervoBib
14	124996	8.928,28571	54328	218	IAutorizacao.GetAplicAutorizadas
13	84824	6.524,92308	13657	952	IDocOcorCurric.AcaoAdaptaCurricAluno
18	79090	4.393,88889	24891	156	ICaixaPostal.GetCxPostalFiltrada
6252	71225	11,39235	14702	0	ISGCA.GetRotulo
418	61101	146,17464	3984	0	IRenovacao.GetRenovacoes
4	59793	14.948,25000	39967	234	IUsuarioGrupo.GetUsuariosNaoGerentes

Fig. 6. Log results obtained for SIE'S RPC.

We analyzed the log data aiming to find optimization opportunities that, for some reason, has been disregarded since the beginnings of the software development (more than 10 years ago) and that may impact the total consumption of system resources. The results showed situations as presented in Fig. 7, which lists the 10 RPC methods with higher total response time. The proportions of the 10 methods (in a total of 602 monitored) in relation to the whole system total response time logged reaches 51.39%.

An analysis of the source code of these method showed situations such as the one of method *IConsultaLocal.ConsultaAcervoBib*, whose optimization would be complex since it has too much possible combinations of parameters and database queries. On the other hand, it also pointed out situations such as the one found on

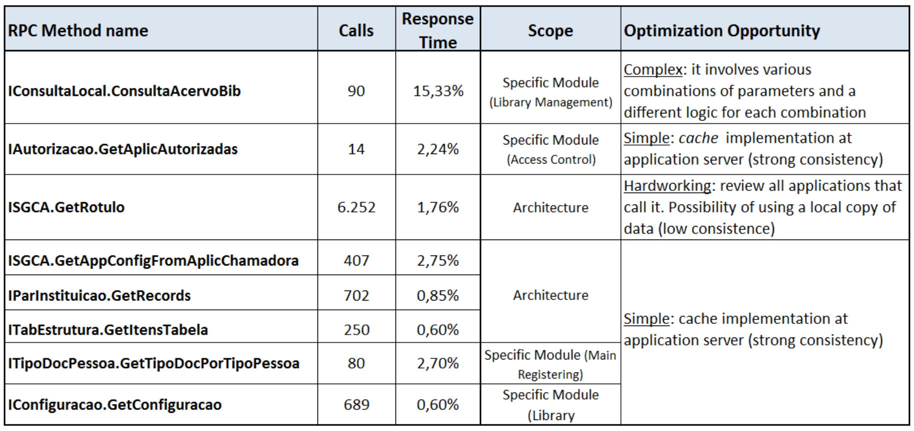
[2] The infrastructure of computers where systems are hosted at UFSM and are accessed by its employees and students.



	RPC method Name	Calls	Total *Response Time*	
			Tempo	%
1	IConsultaLocal.ConsultaAcervoBib	90	385.336,0 msec	19,17%
2	IAutorizacao.GetAplicAutorizadas	14	124.996,0 msec	6,22%
3	IDocOcorCurric.AcaoAdaptaCurricAluno	13	84.824,0 msec	4,22%
4	ICaixaPostal.GetCxPostalFiltrada	18	79.090.0 msec	3.93%
5	ISGCA.GetRotulo	6252	71.225.0 msec	3.54%
6	IRenovacao.GetRenovacoes	418	61.101.0 msec	3.04%
7	IUsuarioGrupo.GetUsuariosNaoGerentes	4	59.793,0 msec	2,97%
8	IMulta.CalculaValorAtraso	220	58.058,0 msec	2,89%
9	IItem.ConsisteRenovacao	34	54.400.0 msec	2.71%
10	ISGCA.GetAppConfigFromAplicChamadora	407	54.134,0 msec	2,69%

Fig. 7. SIE RPC methods with higher total response time (20 min. monitoring).

method *ISGCA.GetRotulo*, which could be optimized by implementing a cache system. An analysis of the whole log data pointed out at least another six methods that could be optimized by a cache implementation. Fig. 8 summarizes that.



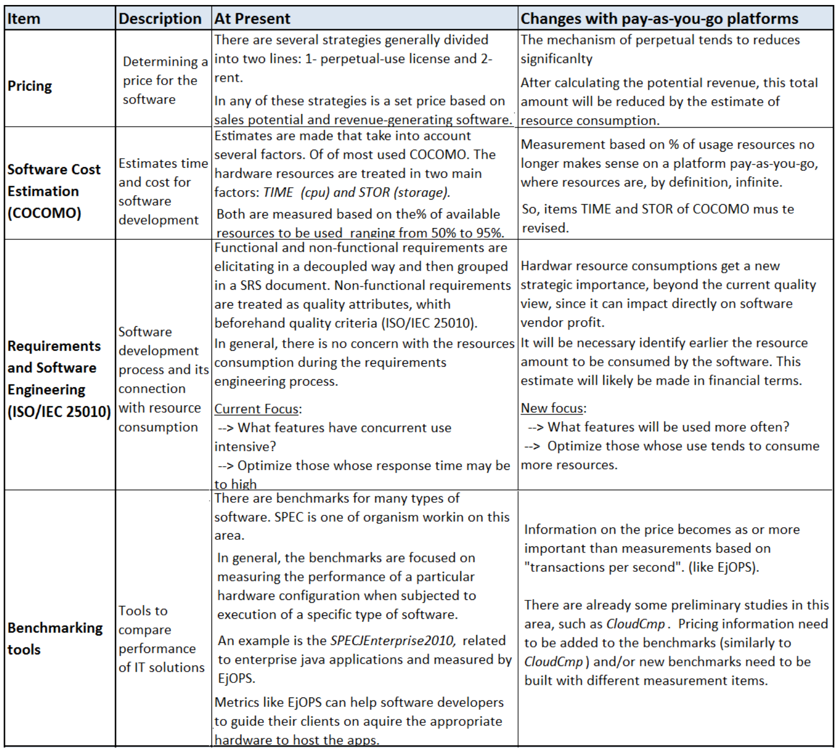
RPC Method name	Calls	Response Time	Scope	Optimization Opportunity
IConsultaLocal.ConsultaAcervoBib	90	15,33%	Specific Module (Library Management)	Complex: it involves various combinations of parameters and a different logic for each combination
IAutorizacao.GetAplicAutorizadas	14	2,24%	Specific Module (Access Control)	Simple: *cache* implementation at application server (strong consistency)
ISGCA.GetRotulo	6.252	1,76%	Architecture	Hardworking: review all applications that call it. Possibility of using a local copy of data (low consistence)
ISGCA.GetAppConfigFromAplicChamadora	407	2,75%	Architecture	Simple: cache implementation at application server (strong consistency)
IParInstituicao.GetRecords	702	0,85%		
ITabEstrutura.GetItensTabela	250	0,60%		
ITipoDocPessoa.GetTipoDocPorTipoPessoa	80	2,70%	Specific Module (Main Registering)	
IConfiguracao.GetConfiguracao	689	0,60%	Specific Module (Library	

Fig. 8. Sample of optimization opportunities found in SIE RPC methods.

A previous study suggests that using a cache strategy within the *AppEngine* platform could be up to 20 times cheaper than using direct access to database [2]. Relying on the connection between *response time* and resource consumption, if we just use a simple cache implementation to optimize these seven methods we could reduce their total response time from 346,536 to 17,326 *msec*. The total response time measured for the whole system during the 20 minutes of monitoring was 2,010,011 *msec*. So it would represent a reduction of 16.38%. This value of 16.38% represents the savings that the software vendor would have with these optimizations if it was using a *pay-as-you-go* cloud platform.

4.3 Discussion

The relationship between *response time* and resources consumption is not fully accurate and the numbers presented in this section cannot be considered definitive. However, it became more evident that a system developed with traditional methods (like SIE ERP was) and without concern for the rational resource consumption can generate wastings that will impact on software vendor profit if the software is deployed over a *pay-as-you-go* cloud platform. The case study points out that the aspects presented in section 3 are really affected by *pay-as-you-go* cloud platform and it will be necessary to revise some of them. In COCOMO II it would be very useful to revise the cost drivers related to *platform* factor. In benchmarking tools, there is a lot of work ahead to meet the new needs related to *pay-as-you-go* model. CloudCmp has begun a good work on that but it will be necessary developing more benchmark suites, similarly to the traditional benchmarks from SPEC. ISO/IEC 25010 and requirements engineering, on other hand, may deserve some minor adjustments in the software requirements process to address concern for rational resource usage. All of these changes are summarized in Fig. 9.



Item	**Description**	**At Present**	**Changes with pay-as-you-go platforms**
Pricing	Determining a price for the software	There are several strategies generally divided into two lines: 1- perpetual-use license and 2- rent. In any of these strategies is a set price based on sales potential and revenue-generating software.	The mechanism of perpetual tends to reduces significanlty After calculating the potential revenue, this total amount will be reduced by the estimate of resource consumption.
Software Cost Estimation (COCOMO)	Estimates time and cost for software development	Estimates are made that take into account several factors. Of of most used COCOMO. The hardware resources are treated in two main factors: *TIME (cpu) and STOR (storage).* Both are measured based on the% of available resources to be used ranging from 50% to 95%.	Measurement based on % of usage resources no longer makes sense on a platform pay-as-you-go, where resources are, by definition, infinite. So, items TIME and STOR of COCOMO mus te revised.
Requirements and Software Engineering (ISO/IEC 25010)	Software development process and its connection with resource consumption	Functional and non-functional requirements are elicitating in a decoupled way and then grouped in a SRS document. Non-functional requirements are treated as quality attributes, whith beforehand quality criteria (ISO/IEC 25010). In general, there is no concern with the resources consumption during the requirements engineering process. Current Focus: --> What features have concurrent use intensive? --> Optimize those whose response time may be to high	Hardwar resource consumptions get a new strategic importance, beyond the current quality view, since it can impact directly on software vendor profit. It will be necessary identify earlier the resource amount to be consumed by the software. This estimate will likely be made in financial terms. New focus: --> What features will be used more often? --> Optimize those whose use tends to consume more resources.
Benchmarking tools	Tools to compare performance of IT solutions	There are benchmarks for many types of software. SPEC is one of organism workin on this area. In general, the benchmarks are focused on measuring the performance of a particular hardware configuration when subjected to execution of a specific type of software. An example is the *SPECJEnterprise2010,* related to enterprise java applications and measured by EjOPS. Metrics like EjOPS can help software developers to guide their clients on aquire the appropriate hardware to host the apps.	Information on the price becomes as or more important than measurements based on "transactions per second". (like EjOPS). There are already some preliminary studies in this area, such as *CloudCmp*. Pricing information need to be added to the benchmarks (similarly to *CloudCmp*) and/or new benchmarks need to be built with different measurement items.

Fig. 9. Summary of changes due to *pay-as-you-go* cloud platform

With these changes in mind, we provide in Fig. 10 a schema that can support further studies that aim to adapt software development process to the new scenario. The presented schema is far way from a model proposal. It merely illustrates the idea of identify the expected usage degree of each feature while taking details about that feature and then estimate the amount of resources that will be consumed to decide if it is significant enough to be designed with the maximum optimization possible. This might support the pricing strategy, since the software price must be huge enough to cover all costs of hardware and generate a satisfactory profit margin.

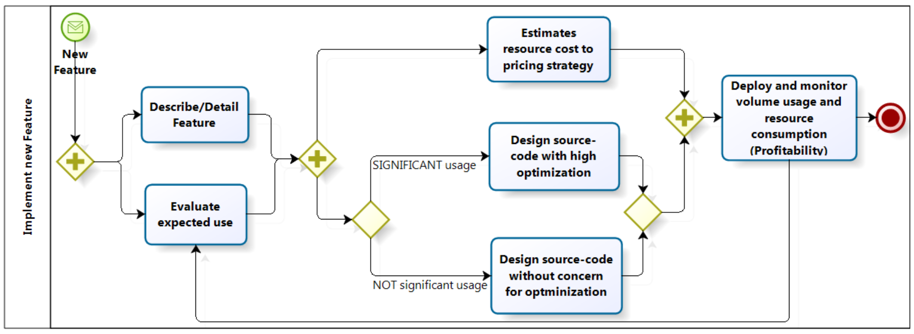

Implement new Feature
New Feature
Describe/Detail Feature
Evaluate expected use
Estimates resource cost to pricing strategy
SIGNIFICANT usage
Design source-code with high optimization
NOT significant usage
Design source-code without concern for optmnization
Deploy and monitor volume usage and resource consumption (Profitability)

Fig. 10. Draft schema to software requirement development process in a *pay-as-you-go* cloud platform

5 Conclusion

Pay-as-you-go cloud platforms represent a new challenge to software developers: they will need to address the hardware resources in a different way than they are used to. This change is related to the fact that, in these platforms, the hardware consumed by the application usage can directly impact software vendor profit. This reality leads to an approach based on rational resource consumption, which is not the focus of traditional software development.

Not all models and tools currently used in software development are ready to deal with this approach based on rational use of resources. The analysis presented in this paper has pointed out the need to review models such as COCOMO II, as well as processes related to requirements engineering. We also identified the need to review *benchmark* suites and algorithms, such as those provided by SPEC.

The case study showed that software systems developed without concern for the rational resource consumption (as ERP SIE) can lead to resource wastes that will impact on software vendor profit if the software would be hosted on a *pay as-you-go* cloud platform. There is still much work to be done in this area and this paper aims to contribute to raise the problems related to rational resource consumption. Further studies may include reviewing other aspects that deserve attention and were not addressed by this work, such as the use of *pay-as-you-go* platform during the software development itself and some pricing issues related to adding new features to existing software in that scenario.

References

1. Barbosa, F.P.: Projeto e implementação de um framework para desenvolvimento de aplicações em três camadas. Tech. rep., Curso de Ciência da Computação. Universidade Federal de Santa Maria., Santa Maria (2000)
2. Barbosa, F.P., Charão, A.: Uma análise do impacto das plataformas pay-as-you-go de computação em nuvem no desenvolvimento e precificação de software. In: Proceedings of the XXXVII Latin American Informatics Conference (XXXVII CLEI) (2011)
3. Boehm, B., Valerdi, R.: Achievements and challenges in Cocomo-based software resource estimation. Software, IEEE 25(5), 74 –83 (sept-oct 2008)
4. Bohem, D.: COCOMO II - model definition manual, version 1.4. Tech. rep., USA (2000), `http://sunset.usc.edu/research/COCOMOII/Docs/modelman.pdf`
5. Dakin, K.: Establishing a fair price for software. Software, IEEE 12(6), 105 –106 (nov 1995)
6. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud computing and grid computing 360-degree compared. In: Grid Computing Environments Workshop, 2008. GCE '08. pp. 1 –10 (nov 2008)
7. Giladi, R., Ahitav, N.: SPEC as a performance evaluation measure. Computer 28(8), 33 –42 (aug 1995)
8. Guha, R., Al-Dabass, D.: Impact of web 2.0 and cloud computing platform on software engineering. In: Electronic System Design (ISED), 2010 International Symposium on. pp. 213 –218 (dec 2010)
9. Hamid R Motahari-Nezhad, Bryan Stephenson, S.S.: Outsourcing business to cloud computing services: Opportunities and challenges. Tech. rep., USA (February 2009), `http://www.hpl.hp.com/techreports/2009/HPL-2009-23.pdf`
10. Harmon, R., Raffo, D., Faulk, S.: Incorporating price sensitivity measurement into the software engineering process. In: Portland International Conference on Management of Engineering and Technology, 2003. PICMET '03. Technology Management for Reshaping the World. pp. 316 – 323 (july 2003)
11. ISO: ISO/IEC 25010. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE). ISO (2011)
12. Kamdar, A., Orsoni, A.: Development of value-based pricing model for software services. In: 11th International Conference on Computer Modelling and Simulation, 2009. UKSIM '09. pp. 299 –304 (march 2009)
13. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp: comparing public cloud providers. In: Proceedings of the 10th annual conference on Internet measurement. pp. 1–14. IMC '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1879141.1879143`
14. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp - pitting cloud against cloud. `http://cloudcmp.net/download` (2011)
15. Pandey, D., Suman, U., Ramani, A.: An effective requirement engineering process model for software development and requirements management. In: International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom), 2010. pp. 287 –291 (oct 2010)

16. Qin, W., Ru-xiang, W.: Research of military software pricing based on binomial tree method. In: 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT), 2010. vol. 9, pp. 628 –632 (july 2010)
17. SEI/PSM: Software quality requirements and evaluation, the ISO 25000 series. `http://www.psmsc.com/Downloads/TWGFeb04/04ZubrowISO25000SWQualityMeasurement.pdf` (2004)
18. SPEC: Standard performance evaluation corporation. `http://www.spec.org/` (2011)
19. Zheng, Y., Cao, R., Sun, W., Zhang, K., Jiang, Z.: Practical application of FDC in software service pricing. In: IEEE International Conference on e-Business Engineering, 2006. ICEBE '06. pp. 352 –357 (oct 2006)