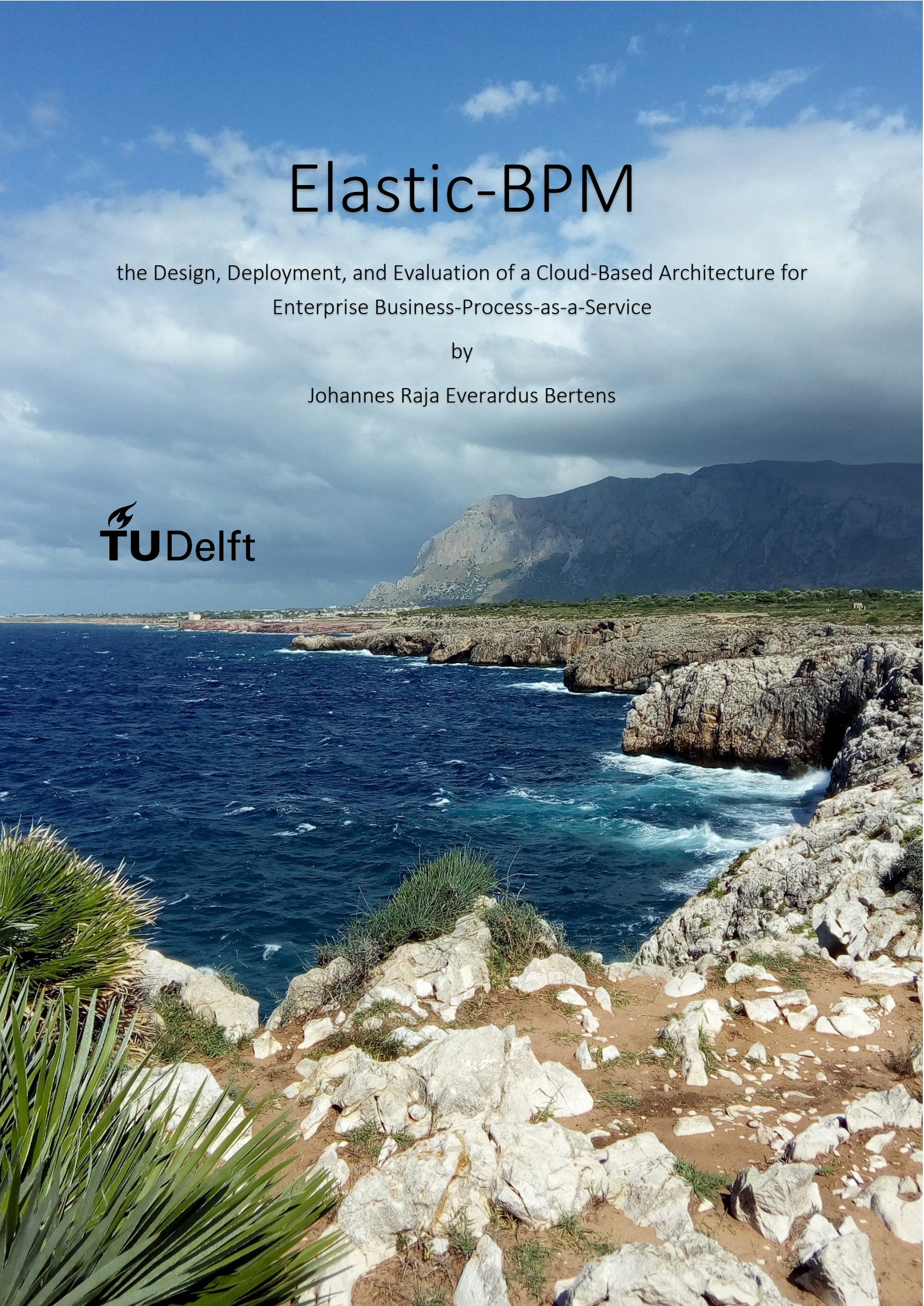


Elastic-BPM

the Design, Deployment, and Evaluation of a Cloud-Based Architecture for
Enterprise Business-Process-as-a-Service

by

Johannes Raja Everardus Bertens



Elastic-BPM

the Design, Deployment, and Evaluation of a Cloud-Based Architecture for
Enterprise Business-Process-as-a-Service

By
J.R.E. Bertens

in partial fulfilment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Friday November 24, 2017 at 10:30.

Supervisor:	Prof. dr. ir. A. Iosup	TU Delft / VU Amsterdam
Thesis committee:	Prof. dr. ir. D.H.J. Epema	TU Delft
	Dr. A.E. Zaidman	TU Delft

This thesis is confidential and cannot be made public until December 31, 2017.

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Last edited: Friday, 10 November 2017



Summary

Business processes are part of every company's daily business. In the past hundreds of years, almost every aspect in the operation of businesses has seen a two-pronged transformation: first, toward defining and then following processes¹, and second, toward making the processes efficient through the use of technology.

In this work, we define a technology-driven approach to business processes: Business Process as a Service. Starting from the current technological possibilities and architectural trends, we define BPaaS, and list requirements for a functional BPaaS system in enterprise settings. Key to our requirements, we consider (1) business process creation, execution, and monitoring, (2) the pay-per-use business model, and (3) useful trade-offs in the cost-scalability space. Using these requirements, we design a reference architecture for BPaaS.

We implement the reference architecture as a prototype BPaaS solution, Elastic-BPM. The key feature of Elastic-BPM is efficient operation, which it achieves by enabling the elastic use of infrastructure. Key components of our reference architecture are (1) the user dashboard and process manager, responsible for the creation, execution, and monitoring of business processes, (2) the virtual infrastructure making use of small, preemptable components, to enable the pay-per-use business model, and (3) the scheduler, responsible for the trade-offs in the cost-scalability space.

The implementation is based on and makes use of state-of-the-art cloud components, such as Angular for the dashboard, Elasticsearch for the scheduler input, and Docker for the virtual infrastructure. We further equip the scheduling component of Elastic-BPM with three different dynamic provisioning algorithms; Static, OnDemand, and Learning, which cover conceptually one static and two dynamic approaches to provisioning infrastructure, respectively.

We validate Elastic-BPM by deploying it both locally and in the Microsoft Azure cloud, and by running over 50 experiments using a synthetic but diverse workload. We find not only optimal setting for the cloud components in the system, but also for scheduling the human resources partaking in the business process.

Acknowledgements

I would like to express my very great appreciation to Alexandru Iosup for the large amount of patience and advice. I would also like to thank my supervisor at Deltares, Pim Witlox, where I executed my earlier work. His humor and critique has been very much appreciated.

Special thanks are given to Rosa Meijer, my better half. She is one of the main reasons this work is completed at all.

Last but not least, I wish to thank my family and friends for all their support throughout these years.

¹ leading to more reliable, more reproducible, and overall more effective operations.

Table of Contents

1.	Introduction	1
1.1	Cloud Computing and Services	1
1.2	BPaaS Requirements	2
1.3	Current SaaS and BPaaS-like Offerings	3
1.4	Research Questions	4
1.5	How To Read This Thesis?	5
2	Current State of IT for Business	6
2.1	Impact of Cloud Computing on Business Models	6
2.2	Software Development Automation and Containers	10
2.3	Architectural Pattern Changes	14
2.4	Current Offering of Business Software	16
2.5	Summary	16
3.	BPaaS Definition	17
3.1.	Formal Definition and Requirements	17
3.2.	Case Studies	18
3.3.	Comparable definitions	24
3.4.	Summary	25
4.	Elastic-BPM: A Reference Architecture for BPaaS	26
4.1.	Requirements for a BPaaS architecture	26
4.2.	Reference Architecture Design	28
4.3.	Functional Components	30
4.4.	Scaling Components	31
4.5.	Infrastructure Components	32
4.6.	Comparable Architectures	32
4.7.	Summary	33
5.	Scheduling in Elastic-BPM	34
5.1.	Metrics	34
5.2.	Processes in BPaaS	35
5.3.	Provisioning resources	36
5.3.1.	Static provisioning policy	37
5.3.2.	On Demand provisioning policy	37
5.3.3.	Learning provisioning policy	38
5.4.	Summary	39
6.	Implementation of the Elastic-BPM Prototype	40
6.1.	The Agile way of working	40

6.2.	The twelve-factor App Methodology	42
6.3.	Creating Microservices for Elastic-BPM.....	43
6.4.	Implementing the components	45
6.5.	Summary	51
7.	Experimental Validation of the Elastic-BPM Prototype.....	52
7.1.	Experiments, workloads and metrics	52
7.2.	Running the experiment	54
7.3.	Results and recommendations	56
7.4.	Summary	66
8.	Conclusion, recommendations, and future work	67
8.1.	Recommendation for building BPaaS applications	68
8.2.	Future work.....	69
	References.....	70

List of figures

Figure 2.1: Most valuable startups in October 2015.	7
Figure 2.2: Illustrating the differences between On-premises, IaaS, PaaS, and SaaS.	9
Figure 2.3: Differences between the use of virtualization and containers for applications.	12
Figure 2.4: Example Dockerfile.	13
Figure 2.5: Grids, Supercomputers and Clouds overview.	14
Figure 2.6: Monolithical applications compared to Microservices applications.	15
Figure 3.1: Student Enrollment case study.	20
Figure 3.2: Online product order case study.	21
Figure 3.3: Purchase reimbursement approval case study.	22
Figure 3.4: Starting NGO case study.	22
Figure 3.5: Customer complaint resolution case study.	23
Figure 4.1: BPaaS Reference Architecture: blue arrows are control flows, yellow arrows are data flows.	29
Figure 6.6: Status changes of workflows and tasks in Elastic-BPM.	35
Figure 5.1: The scheduler component and the components it influences	36
Figure 5.2: Static policy depicted over time for an example workflow.	37
Figure 5.3: On Demand policy depicted over time for an example workflow.	38
Figure 5.4: Feedback Learning policy depicted over time for an example workflow.	39
Figure 6.1: Visualization of technical debt.	41
Figure 6.2: Test Driven Development (TDD) lifecycle.	42
Figure 6.3: Windows and Linux support for Docker.	44
Figure 6.4: Screenshot of a live-updated DataTable.	47
Figure 6.5: Test flow, visualized using Cytoscape.js.	47
Figure 6.7: Output when running the Docker 'node ls' command.	50
Figure 7.1: Arrival times for w1 (top 3) and w2 (bottom 3), subdivided per type of workflow.	53
Figure 7.2: Work schedule ii for simulated humans.	55
Figure 7.3: Absolute runtime of the static algorithm, with a different number of threads per run.	57
Figure 7.4: Relative runtime of the static algorithm, with a different number of threads per run.	57
Figure 7.5: Absolute runtime of the static algorithm, with a different number of machines per run.	58
Figure 7.6: Relative runtime of the static algorithm, with a different number of machines per run.	58
Figure 7.7: Average runtimes for experiment c.	59
Figure 7.8: Relative runtime for experiment c.	59
Figure 7.9: the average load of all active nodes for w1 and w2 during experiment c.	60
Figure 7.10: Average runtime showing the influence of the OnDemand scaling threshold.	61
Figure 7.11: Average runtimes for different scaling thresholds for Learning provisioning policy.	62
Figure 7.12: Average runtimes with different number of humans in the loop (experiment f).	63
Figure 7.13: Average runtime when comparing the different worker schedules.	64
Figure 7.14: Average runtimes of the 10 runs with identical input parameters.	65
Figure 7.15: Relative runtimes of the 10 runs with identical input parameters.	65
Figure 8.1: The evolution of Software Architecture.	68

1. Introduction

Every business has at least a few *business processes* [1]. From the mundane approval processes executed on a day to day basis to the on-boarding processes executed only every now and then to extreme complex logistic challenges within a shipping company. For some companies, there are only a few processes that together make up the bulk of the work executed and value added, other companies have a large diversity of processes that each add value to the company.

These processes all vary in the way which they are automated and executed by computer systems. At small companies, often only a handful of business processes are automated, while at large companies even processes executed only every now and then, like the on-boarding of new hires, are worthwhile to automate at least partially due to the large numbers involved.

Currently, there are a few dozen possible solutions for business process management, ranging from relatively light software packages aimed at small and medium enterprise customers, to large software solutions integrating with a range of enterprise software out of the box with a matching price tag [2], [3]. Current software solutions have one thing in common: they are not built to scale. These *simple* software solutions need to be installed locally and will run fine for years to come, some are even hosted “in the cloud” and are offered as a service. This however, does not grant the full benefits of a Business Process as a Service (BPaaS) solution. The benefits of a true cloud solution include but are not limited to, near-infinite scaling, complete isolation of processes, and a pay-per-use pricing model.

In this first chapter, we explain what Business Process as a Service entails; we give a short background into Cloud computing and services in Section 1.1, in Section 1.2 we lay out the fundamental requirements any BPaaS should adhere to. A look at current offerings of SaaS and BPaaS-like systems will be given in Section 1.3, the research questions for this thesis will be made explicit in Section 1.4 after which the structure of the rest of this thesis will be explained in Section 1.5.

1.1 Cloud Computing and Services

Cloud computing is a computing paradigm under which IT services, be they infrastructure or software, can be leased (provisioned) easily, on-demand (when needed, for as long as needed), and subject to non-trivial terms of service that typically use the pay-per-use pricing model. The popularity of cloud computing is growing rapidlyⁱ, which also leads to a large increase in the amount of services online. Services for social activity, email and hosting blogs or movies are well known – examples are Facebook, Twitter, Gmail, Outlook, WordPress, Blogster, and YouTube. There are also services for infrastructure, networks and databases – these are often used as building blocks for other services. Quite a few startups are using cloud services for their core business. The usage of cloud services gives such startups a large advantage in scaling and flexibility as compared to larger companies and government bodies, as the starting costs for the usage of cloud services are low and they often only pay for what is used.

Services where access to infrastructure is offered are *Infrastructure as a Service* (IaaS). Cloud providers offer IaaS in the form of virtualⁱⁱ or physicalⁱⁱⁱ machines for lease, often billed per hour or minute^{iv}. These machines can then be used for anything traditional hosting has been used for, ranging from web site hosting to large computational grids.

ⁱ This is explained in more detail in Chapter 2

ⁱⁱ Most cloud providers provide virtual machines, for example AWS, Microsoft Azure and Google Compute

ⁱⁱⁱ A good example of a successful physical Cloud machine solution is Packet.com

^{iv} Microsoft Azure and AWS now bill virtual machines per minute instead of per hour

A higher level of abstraction offered by cloud service providers is *Platform as a Service* (PaaS). These are building blocks, created to be used in a combination to create a software solution. Many SaaS offerings can be created using PaaS components. The cloud provider takes care of the underlying operating system and scaling requirements, where possible.

Not all companies require all services offered, most services are mainly geared towards consumers. For example: Twitter, Facebook and YouTube are free of monetary cost and mostly used by consumers. These services are *Software as a Service* (SaaS) solutions. The end-user just makes use of the service, without having to configure much – in some case only the creation of an account or profile is required to fully use the service.

For companies (startups and large enterprises alike), universities, schools, and NGOs, there is a universal need for a cost-effective and powerful solution that helps them in executing and administering their core and supporting *business processes*, which often combine IT with person-to-person interaction. For example, companies that sell items on their website require orchestration to give customers timely updates on how the order is being handled. For companies, universities, schools and NGOs alike there also is a strong need for an onboarding procedure: creating badges and granting access to software systems and facilities. An offering that provides these services can best be described as Business Process as a Service (BPaaS) which we will describe in more detail next.

1.2 BPaaS Requirements

BPaaS is a directly usable service for business processes, where PaaS offerings require configuration and adjustment. The current SaaS offerings for business processes in the market are scarce and expensive, which is in stark contrast when compared to cloud offerings in the IaaS, PaaS and general-SaaS space.

To clearly define the requirements for BPaaS, or business process as a service, we must first define what the business process part of the software requires and what the “as a service” part of the software entails:

- Functional requirements are the **business process** part of the software requirements. They state what the software should do. Examples of functional requirements for business processes are the creation, execution, and monitoring of business processes. These requirements are vital for any business process management software.
- The “**as a service**” part of the software defines the non-functional requirements. Examples of non-functional requirements for business processes are maintaining a consistent state of the running and finished processes at all times, and scaling according to demand.

The potential societal and industrial impact of an available BPaaS solution is high, as this enables small and large businesses, as well as educational institutions and NGOs, to automate their business processes with a low entry cost and pay-per-use billing. This enables starting businesses to use BPaaS for billing, approval, and on-boarding business processes without having to buy expensive, specialized, software for this while growing, reducing their overall IT-spend. It also enables larger corporates to augment current manual tasks with automated business processes, reducing the end-to-end time required for secondary business objectives.

Although BPaaS can have a high societal and industrial impact, there is a lack of commercial BPaaS solutions. One of the main reasons is that, currently, there exists no clear definition of what a BPaaS solution should include. We need a set of minimal requirements for BPaaS. From the few disparate theoretical frameworks and practical approaches that do exist, and from a set of use-cases we introduce in this work, we derive a set of minimal requirements, which we have listed in Chapter 3.

1.3 Current SaaS and BPaaS-like Offerings

Current SaaS cloud offerings including business process tooling include ProcessMakerⁱ, BPM'Onlineⁱⁱ, KissFlowⁱⁱⁱ, and Leankit^{iv}. These services provide business process management, but fall short of the complex requirements of BPaaS.

Larger SaaS offerings, such as SAP^v, Salesforce.com^{vi}, and Dynamics CRM^{vii} suites, also offer business process capabilities. These enterprise packages offer great benefits for large enterprises, however, the setup and maintenance costs (CAPEX and OPEX, respectively) are too high for many smaller companies, universities, schools, and NGOs. In addition, the business process capabilities in these systems are lacking functionality and do not offer a good solution for all business process requirements. We give a comparison of these different systems in Table 1.1. From our analysis, we conclude there is not a single offering with a pay-per-use pricing model. The current models are based on a combination of users (accounts) and different options (basic / standard / premium). Determining the elasticity and how the different options scale have proven to be difficult. ProcessMaker and BPM'Online have comprehensive documentation from which it is clear that there is no automatic scaling possible. From past personal experience, the Dynamics CRM offering also offers only limited scaling possibilities.

	SaaS with extended business process support				General SaaS		
	ProcessMaker	BPM'Online	KissFlow	Leankit	SAP	Salesforce	Dynamics CRM
<i>Cost (yearly)^{viii}</i>	Free/\$12.000+ ^{ix}	\$2.500+	~ \$1.000	\$2.000+	\$4.800++	\$9.000++	\$25.000++
<i>Elasticity</i>	None ^x	None ^{xi}	Unclear	Unclear	Unclear	Unclear	Unclear/limited
<i>Flexibility</i>	5/5	5/5	5/5	5/5	1/5	3/5	3/5

Table 1.1: Comparison of current BPaaS-like offerings available.

For the suites geared towards large enterprises, a true BPaaS solution would be a great addition. Incorporating a true business process solution in the offering could be a great differentiator and unique selling point, while ensuring the service stays scalable. For smaller offerings, like the first ones listed in Table 1.1, a pay-per-use pricing model and automatic scaling will be a benefit for both the consumer and producer of the software.

ⁱ ProcessMaker: processmaker.com/

ⁱⁱ BPM'Online: bpmonline.com/products

ⁱⁱⁱ KissFlow: kissflow.com/

^{iv} Leankit: leankit.com/

^v SAP: go.sap.com/index.html

^{vi} Salesforce: salesforce.com/eu/

^{vii} Dynamics CRM: microsoft.com/en-us/dynamics/crm.aspx

^{viii} Pricing is difficult to find for the general SaaS solution, these have custom prices including the BPM addons

^{ix} Free = limited functionality

^x ProcessMaker server requirements: wiki.processmaker.com/index.php/ProcessMaker_Server_Sizing

^{xi} BPM'Online server requirements: academy.bpmonline.com/documents/marketing/7-8/server-side-system-requirements#XREF_83080_1_1

1.4 Research Questions

In this Section, we describe the main Research Question and underlying research questions this thesis sets out to answer.

Main Research Question

RQ: How to develop a cloud based business process service, such that companies requiring business process management can make use of this service on a pay per use basis, ensuring useful trade-offs in the cost-scalability space?

Research Questions Derived from the Main Research Question

RQ1: How to define a cloud-based service for business-process creation, deployment, and management (BPaaS)?

RQ2: How to create a reference architecture for BPaaS that is cost-effective and scalable?

RQ3: How to equip the BPaaS architecture from RQ2 with provisioning policies that provide useful trade-offs in the cost-scalability space?

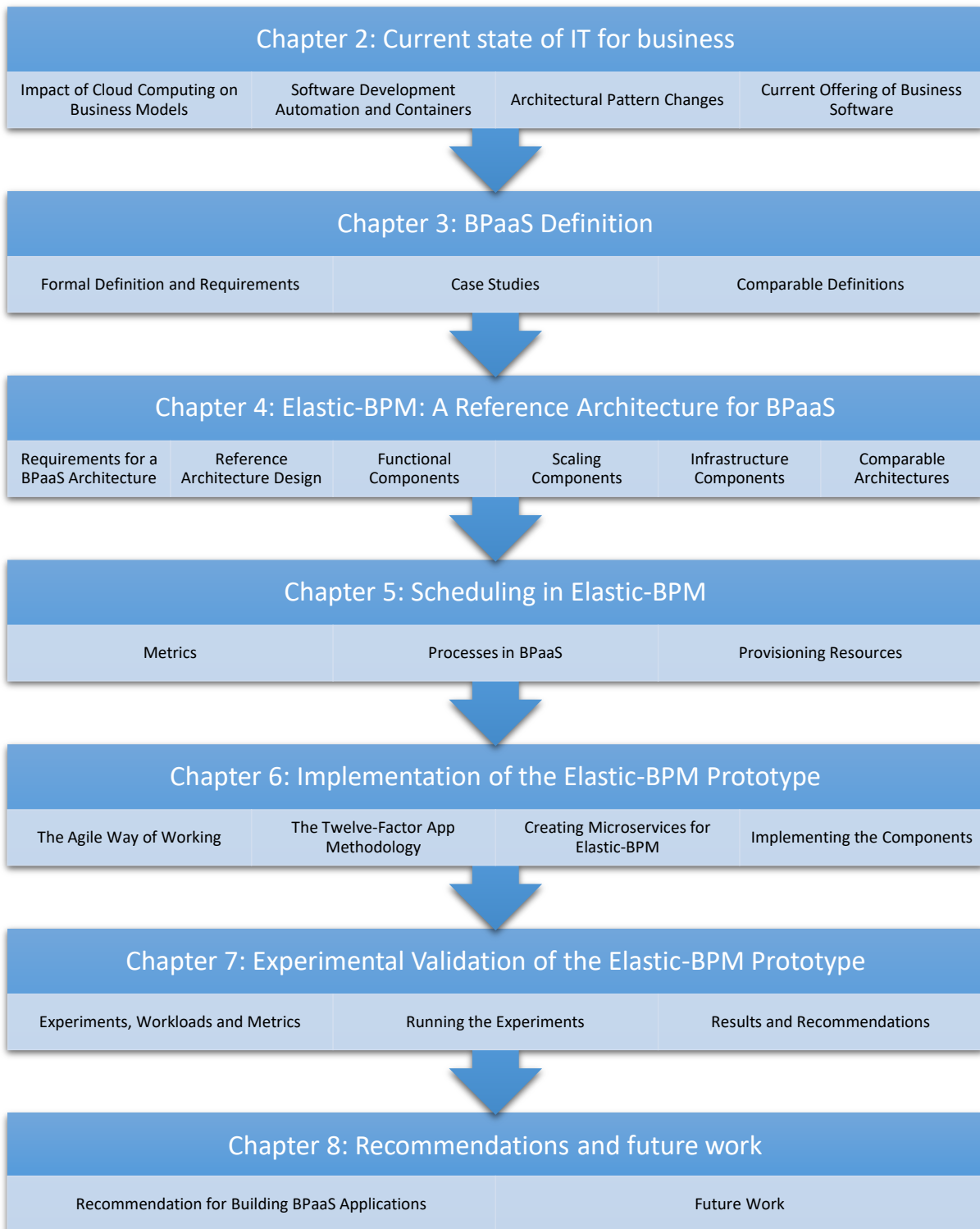
RQ4: How to demonstrate, through a prototype, the characteristics of the BPaaS architecture from RQ2 and of its policies, for realistic conditions?

Answering the research questions derived from the main research question does not fully answer the main research question, but gives a foundation for answering our main research question. By answering them, we ensure as part of the foundation that, respectively: (1) the definition of BPaaS is defined, (2) there is a reference architecture, (3) this reference architecture is equipped with adequate provisioning policies, and (4) a prototype implementing these policies has been demonstrated.

Moreover, the results of the experiments on the prototype implementation could be used as basis for further research and a production-ready implementation of a BPaaS service.

1.5 How to Read This Thesis?

The rest of this thesis is structured as follows:



Chapters 2, 3, 4, 5 and 8 are focused on the theory of modern applications and **why** the BPaaS proposal is so appealing. Chapters 6 and 7 are technical in nature, giving details on **what** is implemented for the system and **how** it performs during real tests.

2 Current State of IT for Business

In the previous Chapter, the Introduction, we have given a brief history on cloud computing and the term used to define different variants of cloud offerings. Afterwards, the term BPaaS has been introduced, including requirements for any BPaaS system. Similar systems have been reviewed and compared, but each failed to fulfill the requirements fully.

To correctly answer our main research questionⁱ, as defined in section 0, we first give a brief overview of the recent history and current state of IT for business. First, we research the impact of compute power and software on business models, next we survey the field of software development automation and containers, we review recent changes to architectural patterns, to conclude with a review of the current offering of business software.

2.1 Impact of Cloud Computing on Business Models

Current smartphones and smartwatches have more compute power and storage than mainframe computers from the previous centuryⁱⁱ, and still there is need for more compute power and storage space. This is due to the demand for computation, the rise of *Big Data*, and other new possibilities as mentioned in [4]. Nearly everything current is in some way connected to a server and a database.

The Jevons Paradoxⁱⁱⁱ, and the more recent Khazzoom-Brookes postulate^{iv}, explain the counter-intuitive relation between supply and demand regarding energy and compute power. More supply of compute power, leads to more consumption and demand. The availability of more data leads to more data-intensive research and demand for data.

UBER, the largest taxi company on the planet, does not own cars. It is founded by two entrepreneurs with a great idea for an app. It runs on servers and databases, not gasoline. Airbnb, the largest hotel chain in the world, does not own a single hotel. It is again founded by two entrepreneurs with a great idea for an app. These companies are no longer the exception, but rather the trend. Figure 2.1 lists the 10 most valuable startups. Uber and Airbnb are only two examples of smart software usage; the list goes on. [5]

Large companies are investing in smart usage of software, instead of acquiring assets. This software needs to run on hardware. Traditionally, large companies would buy computers and put them somewhere on the premise, most probably in the basement or a storage location. This is how the term “on-premises” or “on-prem” originated, in contrast to “off-premises”, which denotes that the servers are not located inside the company but somewhere else.

Grid computing was, and is of still, used by companies and universities that need a large amount of computational power on a regular basis. [6] Grid computing combines the power of multiple, often identical, machines to solve a single task. Once the task is completed the involved machines return to the pool of machines, to be used for the next task. The size of such a pool of machines varies greatly, as does the time between hardware upgrades and costs of single machines. Cloud computing is being used more and more to meet these demands, due to the flexibility and the reduced cost of purchase. [7], [8], [9]

ⁱ “How to develop a cloud based business process service, such that companies requiring business process management can make use of this service on a pay per use basis, ensuring useful trade-offs in the cost-scalability space?”

ⁱⁱ This article from Nature has a great graphic on CPU power: <http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338>

ⁱⁱⁱ Wikipedia: en.wikipedia.org/wiki/Jevons_paradox

^{iv} Wikipedia: en.wikipedia.org/wiki/Khazzoom%E2%80%93Brookes_postulate

Virtualization has made cloud computing possible. Using virtualization, the traditional one-on-one relationship between computer and operating systems is no longer a limiting factor. This leads to several benefits; including but not limited to *Server consolidation*, *Disaster Recovery*, *Load balancing* and *Virtual desktops*.

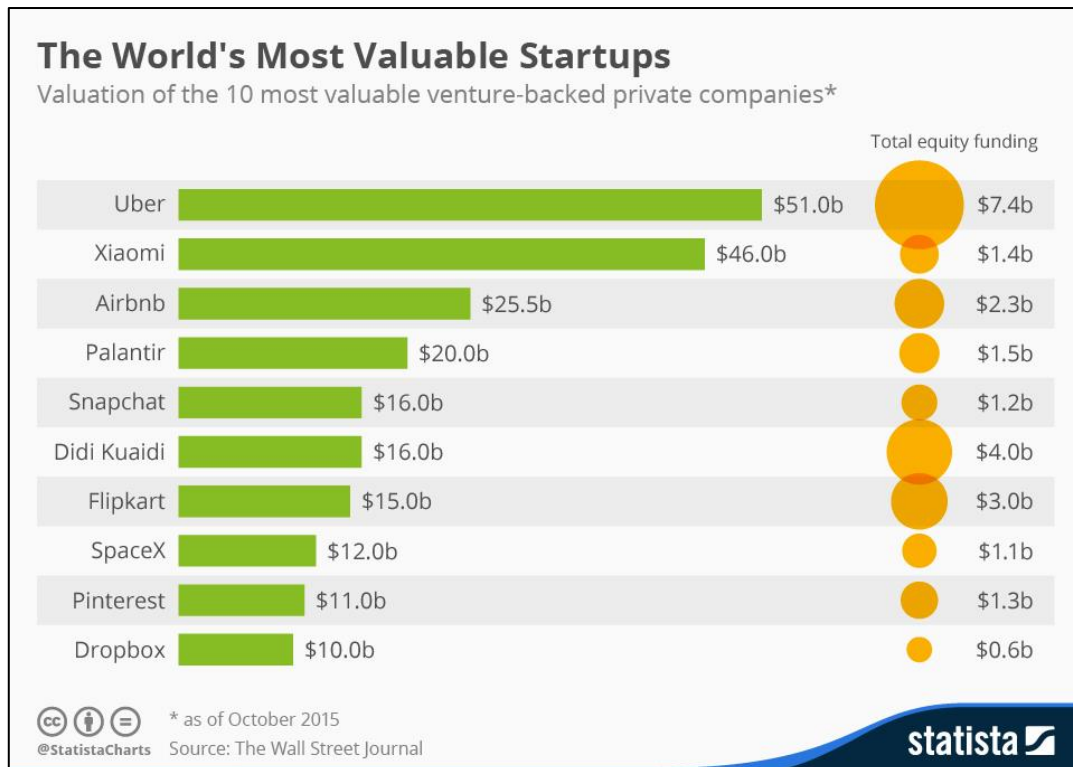


Figure 2.1: Most valuable startups in October 2015.

Sever consolidation is a benefit that is being used by every major Cloud provider. Especially taking into account the shared-CPU models often used as budget option, the provider is then free to *overbook* physical hardware with virtual machines. This cost-effective solution is great for web-hosting solutions and other scenarios where the server is used in bursts. This solution is not desirable in a situation where all *tenants* are running heavy-load tasks on the machines, which will lead to an overall bad and unpredictable performance for every executed task.ⁱ

Disaster Recovery (DR)ⁱⁱ and the creation of backups is made easy by virtualization due to the fact that the hard disk is also virtualized. Backups (also called snapshots) can often be made while the system is running without the user noticing any change in performance. When these backups are stored on a separate machine or location, they can be used in the case of a disaster on different hardware within minutes or seconds instead of the hours or days as is often the case without the use of virtualization. With the use of Cloud computing, the cost of DR is also reduced significantly. [10], [11]

Load balancing is not new for virtualization. It has been used with physical hardware before and will continue to be a large part of the internet-facing websites for years to come. The biggest feat virtualization adds, is the ability to scale the number of machines backing the load balancer on demand. This is a major benefit for websites or applications regarding time-bound events. Examples could be the world cup for any

ⁱ Server Consolidation background information: vmware.com/nl/solutions/consolidation.html

ⁱⁱ Disaster Recovery background information: en.wikipedia.org/wiki/Disaster_recovery

major sport or the Olympic games. News sites could also benefit largely scaling out and in when required. [12], [13], [14], [15]

Virtual desktops have been gaining and losing popularity over time the last decades.ⁱ Citrix is well known supplier in this regard of virtualization. Traditionally, virtual desktop infrastructure has been implemented using servers hosted on premises or at a hosting company. This gives the IT-support the advantage of centralized management, improved data security and simplified deployment. The costs have been a large detractor for this solution: there need to be enough physical machines for each user to use his or her virtual desktop at the same time. While sharing physical hardware between users (see server consolidation) is an added benefit of virtualization, this only goes so far. The advent of cloud computing alleviates this ailment. Moving towards a “Desktop as a Service”ⁱⁱ offering might give virtual desktops the boost it needed to become well adapted in the market. This is however outside of the scope of this thesis.

Cloud computing has become quite popular, with almost all large companies offering computational power for hire. The concept is not new; it has been noted in the 1960s by John McCarthy that “computation may someday be organized as a public utility”. This is also where the term for grid computing finds its origins. However, where grid computing has failed to live up to the promise, cloud computing might just bridge this gap. [16], [17], [18],

The most straightforward use of cloud computing is the form where virtual machines are rented from a third-party on a pay-per-use basis. The virtual machines are started when required and shut down when the task is finished. [19], [20] In this scenario, the user does not own the infrastructure executing the task, but leases it. There are still scenarios where the best solution is to buy and own the hardware instead of using a cloud solution. Ill-informed companies sometimes expect a large saving of costs by moving the current infrastructure and software to a cloud provider, without investing in a structural improvement and redesign. This practice is called *lift and shift*. [21], [22]

The direct usage of infrastructure rented from a cloud provider, is called *Infrastructure as a Service*. There are other Cloud models which, depending on the scenario, can yield a lot more improvement in regards to scaling and reduced cost to serve. [23] These models are called *Platform as a Service* and *Software as a Service*. In Figure 2.2 these definitions and the baseline *on premises* (or *your own computer*) are shown in a side by side comparison. Here the light-colored blocks are parts of the system you are responsible for and the dark-colored blocks are the responsibility of the cloud provider.

Buying server hardware and placing it on the company premises, gives you the largest possible freedom, but also comes with all of the maintenance burden. The company then needs to make sure the network is stable, the storage is backed up, the servers are upgraded to make sure the latest vulnerabilities are patched, the virtualization (if used) is installed and configured correctly, and more. For one or two machines, this might not sound like a lot of work, but when the company grows or the servers need to be upgraded, these chores tend to add up to quite a lot of work and investment. [24]

ⁱ Desktop Virtualization background information: en.wikipedia.org/wiki/Desktop_virtualization

ⁱⁱ Introduction to DaaS: questsys.com/desktops-as-a-service/what-is-daas.aspx

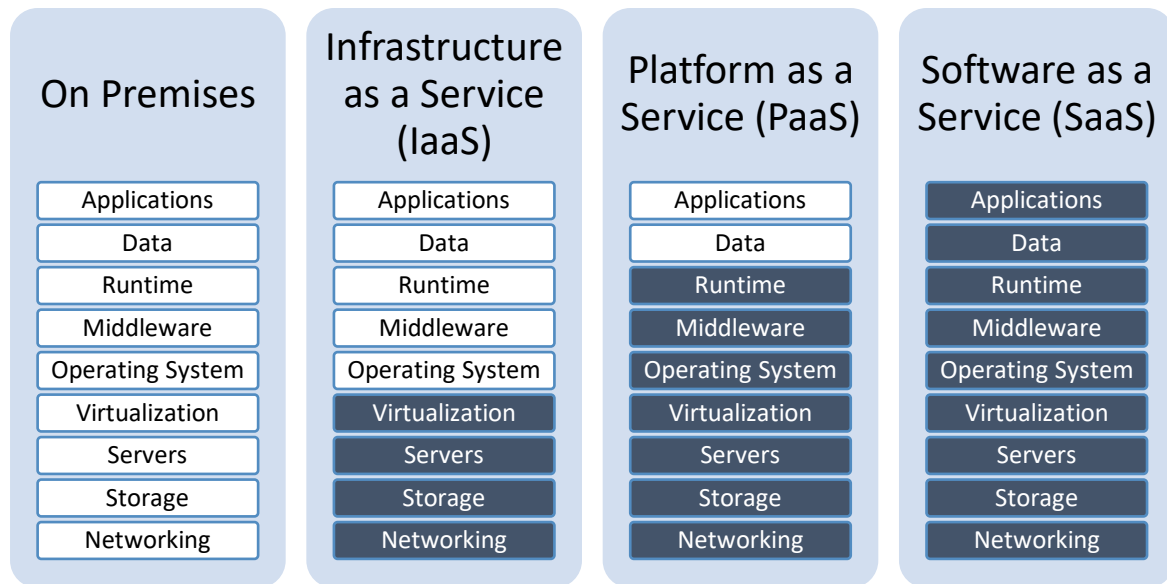


Figure 2.2: Illustrating the differences between On-premises, IaaS, PaaS, and SaaS.

One level higher on the scale of relinquishing control, is the *Infrastructure as a Service* or IaaS option. [24] When using IaaS, the network, storage, server hardware and virtualization software is maintained by the cloud provider. The operating system and other software on the machine is still your own responsibility. This still gives a large degree of freedom: there is choice between Windows or Linux, you can choose which services are installed, what middleware is used and the choice of runtime platform is also up to you.

IaaS seems like a great solution for most businesses working with specialized software, a great trade-off between freedom of choice and maintenance costs. In practice, however, the operating system, services, and runtime used often does not play a crucial role in software development. A great example is hosting a website created with JavaServer Pages (JSP)ⁱ. Whether the JSP server is run on a Linux or Windows host, does not matter for the websiteⁱⁱ. It also does not matter how the logs are collected, only that they are accessible. The same goes for websites created with other host-agnostic tools like Python, PHP, NodeJS, and recently even ASP.NETⁱⁱⁱ. As long as the website works as required and can be updated by the developers, the operating system used and services running are not relevant.

The next step on the cloud-scale is called *Platform as a Service* or PaaS. With this option of cloud service, the user is only responsible for the data and application itself. In practice, this often means uploading the source code to a specific location or using the platform SDK and taking care that the data is stored in an accessible location. This could be a local file or database, or a database or service hosted externally. The specifics vary from provider to provider. Universal in all PaaS configurations is that the cloud provider takes care of everything up to and including the runtime. This means that when you are using PaaS components for a Python website, the cloud provider takes care of keeping the Python runtime up to date, including the underlying operating system and services and the hardware. [25]

The highest level on this scale is called *Software as a Service* or SaaS. This model is used for software that is not the primary focus of the company itself, but rather facilitating the workforce. Good examples are software to make sure employees are paid a certain amount every month, software to manage the stock of a retail company, and software to create and share documents. For these situations, there are services

ⁱ An introduction to JavaServer Pages: en.wikipedia.org/wiki/JavaServer_Pages

ⁱⁱ There might be exceptions, due to the way Java is implemented on either Windows or Linux

ⁱⁱⁱ Hosting ASP.NET on linux: docs.microsoft.com/en-us/aspnet/core/publishing/linuxproduction

that can be used online. Examples are SAP, Salesforce or Microsoft Office 365. There are also other, smaller, SaaS solutions, for example for creating and updating a to-do list there are multiple online solutions: Todoist, Trello, Wunderlist, and “Remember the Milk”, are only a small section of these solutions. The amount of SaaS offerings grows in a rapid pace.

The use of SaaS has been rising quite a while now. [26], [27] Facebook and Gmail are great examples in this field, but not the first. This business model is also spreading to businesses not offering software, but deal with tangible items. A great example is the Dutch lighting and electronics concern Philips. Philips is now offering *Lumen as a Service* for new building projects and companies. This way of business is also possible with art. There are a few companies offering *Art as a Service*, for example the *Kunsttuitleen* (Art library) in large cities. Most recently there is a new company from Delft offering *Bikes as a Service* called *Swapfiets*ⁱ. Traditional libraries can also be defined as offering *Books as a Service*.

Software as a Service is often the best option for facilitating software [28], but not for the core competency of the business. For the core competency or focus of the company you often need to customize the software more than what is possible using a *SaaS* solution. For these situations, the best option could be to use a *Platform as a Service* solution. Using a *PaaS* solution makes sure the company can focus on what differentiates them from the competition. Focus on the application and the data rather than the required software and hardware to run this application. For example: most websites are easy to create using a *PaaS* solution. Most relational databases are also readily available as *PaaS* solution, this is also often called a *SaaS* or *Database as a Service (DBaaS)* solution.

For most businesses, *PaaS* solutions offered by Cloud providers do not cover all required elements. For example, most NoSQL database solutions are not yet available as *PaaS* solution. Also, specific runtimes or log collection services, like Elastic or Statd, are not yet available as *PaaS* solution on all providersⁱⁱ. Resorting in these cases to *Infrastructure as a Service*, even though this seems like a great solution, can lead to saturation of these virtual servers, which can lead to a new server park, but now in the cloud. These servers also require maintenance and troubleshooting, which detracts from the main focus of the company.

In situations where not all software can be used as *SaaS* and not all required components can be run in a *PaaS* solution, an alternative can be to use a *Containers as a Service (CaaS)*. Most large cloud providers now also provide this option; however, these solutions are often still unproven and might not be suited for production work. This is a space where research is still being done. [29], [30], [31], [32] The use of containers for software development is explained in more detail in the next section.

2.2 Software Development Automation and Containers

When a new Virtual Machine (VM) is created, this is all done in software, and takes only minutes instead of hours or days, as there is no need for the placement of physical components and wiring for connectivity. Physical hardware is still required of course, but due to virtualization can now be viewed as a commodity rather than an intrinsic part of the system. Due to this view on physical hardware, it is becoming increasingly common to automate the creation, installation, and configuration of VMs.[33]

A large added benefit of the automation, is the repeatability. If everything is executed according to an automated script, it can be repeated exactly without change. The value of this automation becomes clear when the necessity of scaling comes into play. The possibility to just delete a machine and create a fresh

ⁱ Swapfiets homepage: swapfiets.nl/en/

ⁱⁱ The offering in *PaaS* and *SaaS* products changes rapidly, it is possible that at the time of reading the mentioned examples *are* available.

environment saves companies a great amount of time in the development process, as running a new test on a clean system has suddenly become trivial.[34]

Automating the creation of VMs in a cloud environment is possible by using the APIⁱ or SDKⁱⁱ from the cloud providers. Individual developers and operations specialists can automate the operations for creating, reading, updating and deleting (CRUD) of the virtual resources.

Using the cloud vendor specific API or SDK, is beneficial only when only a single cloud provider used. When consuming services from different cloud providers, the benefits diminish, as scripts, and added support for new features can take up more time than it saves. In these scenarios, there are quite a few cloud middleware API providers that can speed up this process.

A selection of middleware API providers are listed in Table 2.1.

LIBRARY NAME	SPONSORING ORGANIZATION	PROGRAMMING LANGUAGE
JCLOUDS ⁱⁱⁱ	Apache ^{iv}	Java
LIBCLOUD ^v	Apache	Python
FOG ^{vi}	independent	Ruby
PKGCLLOUD ^{vii}	independent	NodeJS

Table 2.1: Cloud middleware libraries.

There are more libraries available and new libraries are being introduced regularly, this is an active field of research. [35], [36], [37]

Vagrant^{viii} is a useful tool for development work and testing purposes, it aids in setting up one or more virtual machines locally. Vagrant gives developers the ability to automate the creation and destruction of an environment consisting of one or more virtual machines on an underlying VMware^{ix}, VirtualBox^x or Hyper-V^{xi} provisioning system. [38],[33]

After the VMs are created and the operating system is installed, other required components for the development, testing, or production environment need to be installed and configured. This installation and configuration is traditionally done by logging into the machine, either using a remote tool or locally and installing these components manually. At most larger companies and universities, this is still the general way of working. Using cloud infrastructure, with machines available within minutes instead of weeks and the number of machines steadily rising, the method of installing and configuring machines is quickly changing from manual to automatic. For the automation of the deployment and configuration of components, there are quite a few mature solutions available today. [39] A recent comparison of the automation tools can be found online^{xii}.

ⁱ API, Application programming interface: en.wikipedia.org/wiki/Application_programming_interface

ⁱⁱ SDK, Software development kit: en.wikipedia.org/wiki/Software_development_kit

ⁱⁱⁱ Apache page for jclouds: jclouds.apache.org/

^{iv} Apache, also known as the Apache Software Foundation, is most known for the http server (daemon), but now supports a wide range of open-source projects.

^v Apache page for libcloud: libcloud.apache.org/

^{vi} Fog homepage: fog.io/

^{vii} Github repository for pkgcloud: github.com/pkgcloud/pkgcloud

^{viii} Vagrant Homepage: vagrantup.com/

^{ix} VMware Homepage: vmware.com/

^x VirtualBox Homepage: virtualbox.org/

^{xi} Hyper-V Documentation: microsoft.com/en-us/server-cloud/solutions/virtualization.aspx

^{xii} A recent comparison of automation tools: blog.takipi.com/deployment-management-tools-chef-vs-puppet-vs-ansible-vs-saltstack-vs-fabric/

A more recent approach to running software, is to run it wrapped in a container. Interestingly enough, **Linux** containers (LXC) have been around since 2008ⁱ. LXC Containers are used to run multiple isolated Linux guest systems on a single Linux host machine. This is comparable to the functionality of virtual machines, but without the need and overhead of actually starting a virtual machine itself. Containers have been gaining popularity steadily since 2015ⁱⁱ. [40]

Figure 2.3 shows the difference in the number of layers required between the application and the server hardware when comparing Containers to the traditional VMs. In this figure, the configuration of the server stack for containers does not use any virtualization between the server hardware and the host OS. In the most used scenario there is still one layer of virtualization at the cloud server provider. This does leave the user with the benefit of lightweight containers. [41], [42]

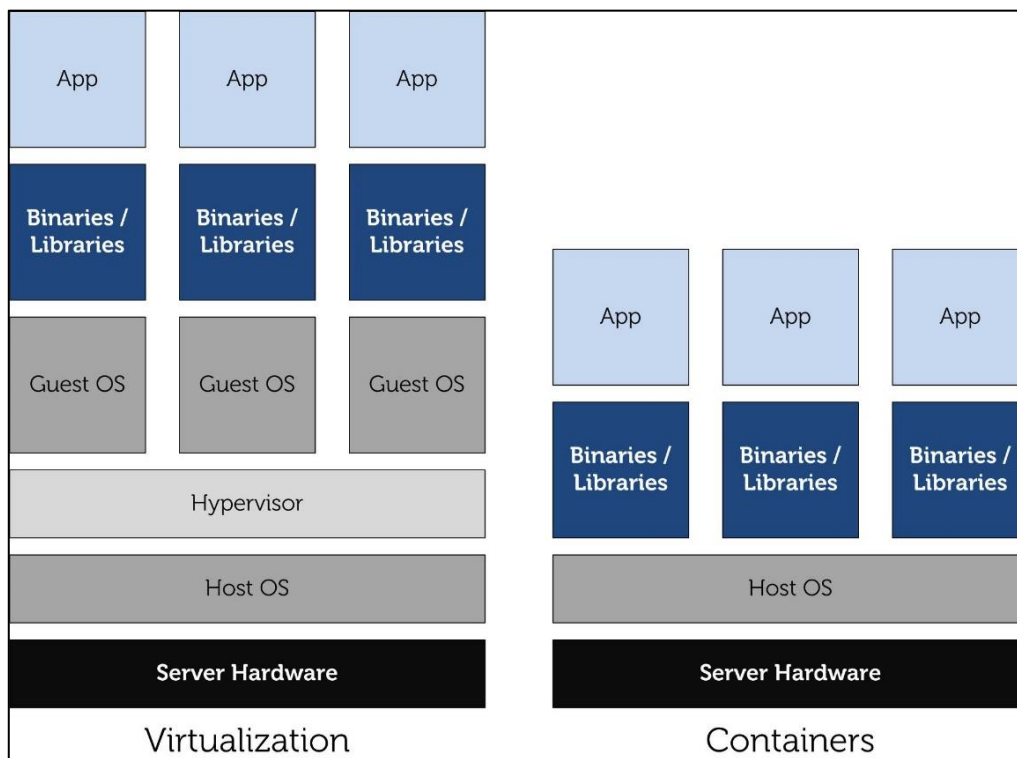


Figure 2.3: Differences between the use of virtualization and containers for applications.

Of all the container implementations, Docker has been gaining the most traction. Docker was first released in the beginning of 2013 and is now adopted at all large Cloud providers (AWS, Google Cloud Platform and Azure) and in addition to running natively on Linux, Windows Server will support Docker in Windows from version 2016 onward. [44]

Docker creates a lightweight and accessible way for developers to build, deploy (ship) and run their applications anywhere in a reproducible fashion. This can also be used to improve the reproducibility in research. [45] Docker introduced the layered approach within Containers: It is possible to build upon a

ⁱ Wikipedia on LXC: en.wikipedia.org/wiki/LXC

ⁱⁱ Popularity rise of Docker and containers: businesscloudnews.com/2016/02/11/exponential-docker-usage-shows-container-popularity/

previously created Docker container, to drastically reduce development and build time. Docker implicitly creates a new image for every step in the Dockerfile, to reduce build and download time.

The Docker system works with a *Dockerfile*ⁱ, which describes which base Docker container is used and how to build and run the Docker container. After building the container using the Dockerfile, it can be run on any Docker host either on-premises or in the Cloud. The Dockerfile used for some of the components of the case-study is shown in Figure 2.4. Combined with NodeJS source code for the application, Docker can use this to build a Docker image that can be run anywhere Linux Docker Containers are supportedⁱⁱ.

```
1 # Base on latest node
2 FROM node:latest
3
4 # Install app dependencies
5 COPY package.json /src/package.json
6 RUN cd /src; npm install
7
8 # Bundle app source
9 COPY . /src
10
11 # Set working dir and run script
12 WORKDIR "/src"
13 CMD ["npm", "start"]
```

Figure 2.4: Example Dockerfile.

The FROM command (line 2) is used to define the base layer to build the new docker image on. In this case, the base layer is one which includes a runtime for NodeJS, called 'node'ⁱⁱⁱ. The trailing ':latest' after the name of the base layer indicates that the latest version of this will be used. It is also possible to define a specific version, for reproducibility. The COPY command (lines 5 and 9) copies files or directories from the development machine into the newly created docker image. In this case we are first copying the list of packages used in our application (line 5), and later we copy the rest of the application into the docker image (line 9). The RUN command (line 6) executes the following statement or statements in the docker image. WORKDIR (line 12) is used to define the location to execute commands during runtime. CMD (line 13) is used to define the commands used during runtime. In this Dockerfile, we define the '/src' directory to be used as working location and 'npm start' as command to execute when the container is started.

With Docker Cloud, Docker takes their container concept and introduces the notion of stacks, services, containers, and nodes. A single container combined with a unique set of environment variables and other configuration settings becomes a service. One or more services together becomes a stack which can be defined using a single Stackfile^{iv}. Each service spawns one or more containers, the desired number of identical containers is configurable per service, which are run on nodes. The nodes can either be created by the Docker Cloud, on any compatible cloud provider, or provided yourself.

ⁱ Dockerfile reference: docs.docker.com/engine/reference/builder/

ⁱⁱ For now, the base image dictates whether the container can be run on Linux or Windows. This might change in the future, depending on the implementation of Docker on Windows.

ⁱⁱⁱ More information on the Node base layer can be found here: store.docker.com/images/node

^{iv} Stackfile reference: docs.docker.com/docker-cloud/apps/stack-yaml-reference/

2.3 Architectural Pattern Changes

A comparison between Cloud Computing and Grid Computing is given in [16], especially Figure 2.5 with Supercomputers, Grids, Clusters, Clouds and “Web 2.0” gives us a lot of information on shifts in architectural patterns between the years 2000 and 2010. The focus here is the overlap and differences between Grids and Clouds, with the x-dimension of this figure depicting “Application Oriented vs Services Oriented”.

In contrast to traditional applications, new applications are more often than not centered around services. Currently a large amount of new applications are online websites or geared towards one or more mobile platforms. Mobile usage has increased tremendously over the last few years. According to recent analysis, this will only increase: the percentage of mobile visits to websites has also risen steadily and will probably surpass the non-mobile visits soonⁱ. Users are growing to expect the same functionality using the phone, tablet or computer for applications and websites. This can be achieved by creating a dedicated application for each platform: mobile, tablet and website, or by creating a set of services that work together to create one universal application.

Setting up an application as a collection of services has the added benefit of being able to maintain easier than one monolithic application. This is also an added benefit for more traditional applications, for example applications that are used internally in a company to administer sales or customer relations, or applications used for research. Although a layered approach to designing applications helps for the maintainability, there are often cross-cutting concerns. Changing the functionality of one layer without influencing another is often difficult or impossible.

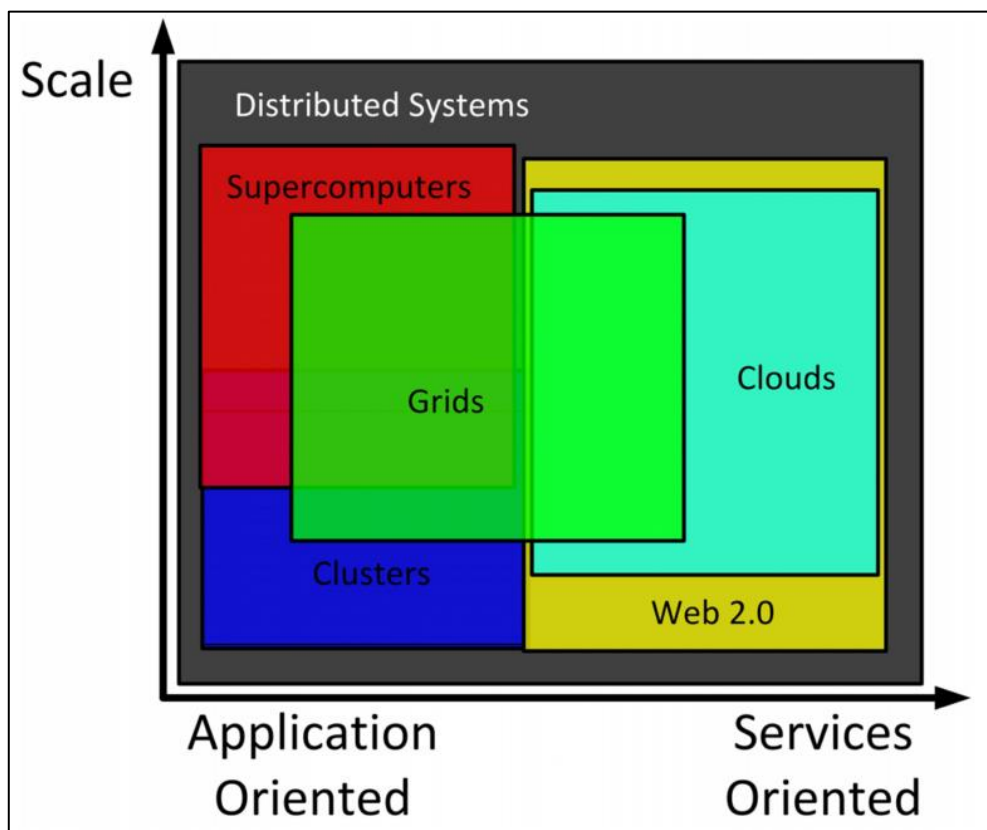


Figure 2.5: Grids, Supercomputers and Clouds overview.

ⁱ Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities: gartner.com/newsroom/id/2939217

Service-Oriented architecture (SOA) or Service-Oriented computing [27] introduces a strict approach to differentiation between layers.[43] Erik Townsend wrote a great introduction on the term in his article with the title “The 25-year history of service oriented architecture”.ⁱ He argues the roots of SOA lay much deeper than is currently believed by most and clarifies the parallels with distributed systems.

The usage of services has increased the last decade. By creating every component or service small enough, but not too small, upgrades and changes are less difficult than they used to be. It is easier to change a small component than to change a large component. This architectural trend, microservices, has seen increased use since 2014.ⁱⁱ Figure 2.6 shows the high-level differences between traditional, monolithic, architecture and microservices architecture, created by Martin Fowler and James Lewis in 2014. Containers technology speed up the adoption of microservice architecture by aiding the creation, building, and, deploying of services. [44]

Important research topics for microservice architecture are: integration, storage, and logging. Different models of microservice architecture have been published, this is an active research topic. [45][46][47]

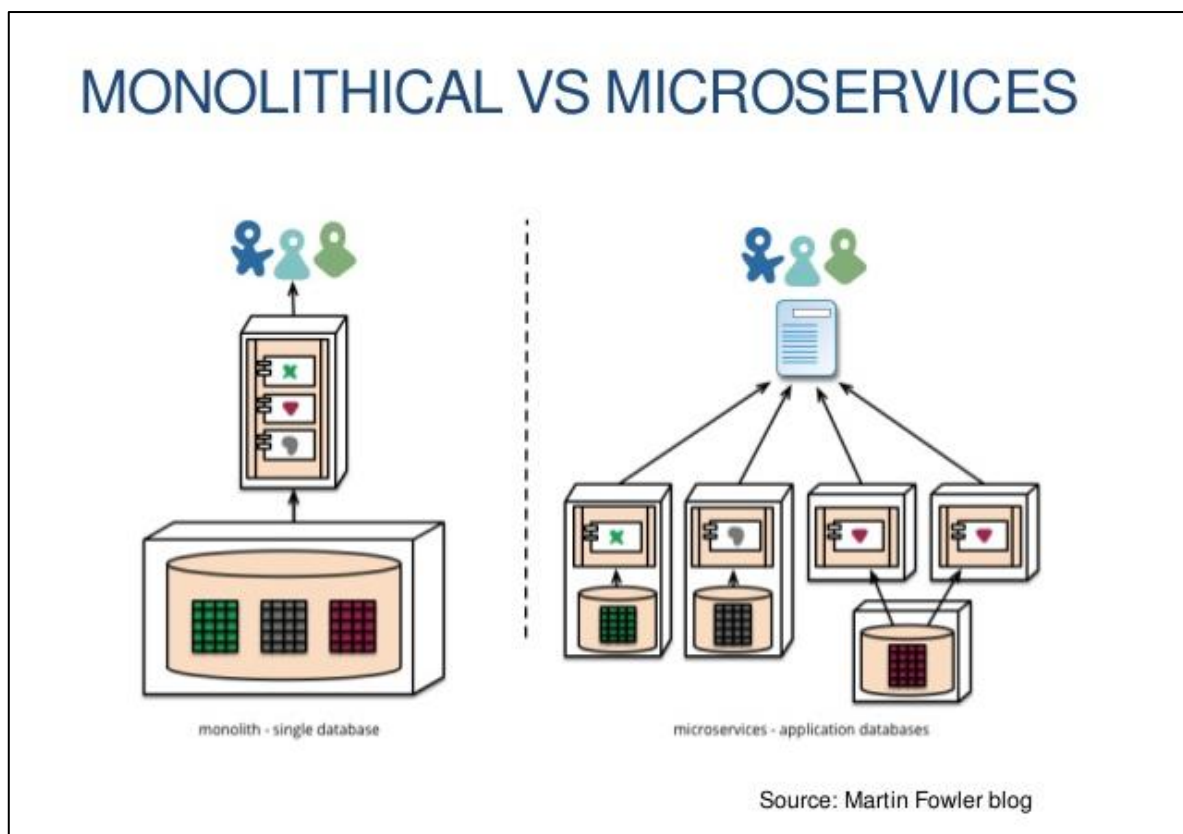


Figure 2.6: Monolithical applications compared to Microservices applications.

ⁱ History of SOA Architecture, by Erik Townsend: eriktownsend.com/look/doc_download/12-1-the-25-year-history-of-service-oriented-architecture

ⁱⁱ Repository of microservices resources, by Martin Fowler: martinfowler.com/microservices/

2.4 Current Offering of Business Software

Business software is quickly changing from installed software with a license fee per processor core, for example the Oracle modelⁱ, towards an online service with a pay-per-user fee, which Salesforce.com has made popularⁱⁱ. Other large software vendors are following, with notable examples being Google's 'G suite'ⁱⁱⁱ and Microsoft with most of their current business software^{iv}.

The pay-per-user, pay-per-GB, and pay-per-minute pricing models have made business software more accessible for small and medium enterprises, for which the price of investing in the hardware required to run these software packages is too high. Data migration between SaaS software providers is more difficult than migrating between traditional software packages, as the access to the underlying database is restricted. This can lead to unforeseen costs when the provider increases the tariffs.

While some of the larger SaaS offerings also provide the ability to define, run, and monitor business processes, the implementation is not flexible enough to fulfill the BPaaS requirements as presented in section 1.2. The current SaaS solutions offering process management, do not offer any cost-effective scalability. A comparison of SaaS solutions is given in section 1.3.

2.5 Summary

In this Chapter, we first explore the impact of cloud computing on business models. Virtualization, especially when combined with a pay-per-use cost model, has led to compute power coming close to a commodity good. The general availability of compute power has made new business models possible. Most of the new large companies are built around software instead of traditional products.

Next, we have examined the reasons behind the automation of development environments. The biggest benefit from this automation, is the ability to scale. Combining automation with Virtual Machines and container technology, has led to a change in architectural patterns. The traditional multi-layered architectural patterns are being replaced by SOA and microservice architectures.

Replacing traditional multi-layered architectural patterns with SOA, and especially microservice architectures, has led to a shift in business models and an increase of SaaS offerings. There is currently no comprehensive BPaaS solution available that scales cost-effectively.

ⁱ The Oracle price list online: oracle.com/us/corporate/pricing/price-lists/index.html

ⁱⁱ The Salesforce.com pricing model: salesforce.com/eu/editions-pricing/sales-cloud/

ⁱⁱⁱ Google's 'G suite' pricing model: gsuite.google.com/pricing.html

^{iv} Microsoft's 'Office 365' pricing model: microsoft.com/en-us/store/b/office365

3. BPaaS Definition

In this chapter, we define what we call Business Processes as a Service, or BPaaS. IaaS, PaaS and SaaS have been discussed in great lengths in the previous chapters, there are many different *-as-a-Service definitions available online, for example: Database as a Service (DBaaS)^{i,ii}, Storage as a Serviceⁱⁱⁱ, Music as a Service^{iv}, and, Integration Platform as a Service (iPaaS)^v. BPaaS has some definitions online as well^{vi,vii}, however, these definitions fail to describe real-life case studies, and lack formal definition.

Our conceptual contribution in this chapter is the formal definition and requirements of BPaaS in section 3.1, we contribute to the library of use cases for BPaaS in section 3.2, after which we compare our BPaaS definition with comparable definitions and give a summary of the chapter.

3.1. Formal Definition and Requirements

We gave a short introduction to BPaaS in the first chapter of this work, to illustrate the need for a BPaaS solution in the industry and non-profit market. To create a reference architecture and implement a working system that fulfills the requirements for BPaaS, we need to first formally define BPaaS, and define the requirements for a BPaaS solution.

Formal definition for Business Process as a Service

Any BPaaS solution, is a solution that offers the ability to visualize, execute, interact with business processes while cost-effectively making use of human and machine resources. The pricing model is a form of pay-per-use.

Requirements for BPaaS

The requirements have been listed briefly in section 1.2, here we describe them in more detail.

- **Ability to execute business processes.**
The core of the system are the business processes – which can be defined as “a set of one or more linked procedures or activities which collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional roles and relationships”^{viii}. Systems adhering to the BPaaS definition must implement a means to administer and execute these business processes. Without this requirement, there is no added value for any business to use this system.
- **Ability to visualize progress in business processes.**
As business processes are often long-running processes, especially when humans are involved (“in the loop”), some form of visualization is required. This is also beneficial to interacting and

ⁱ Introduction to DBaaS: techopedia.com/definition/29431/database-as-a-service-dbaas

ⁱⁱ Oracle’s take on DBaaS: oracle.com/technetwork/oem/cloud-mgmt/dbaas-overview-wp-1915651.pdf

ⁱⁱⁱ Introduction to Storage as a Service: techopedia.com/definition/24900/storage-as-a-service-saas

^{iv} Article: aisel.aisnet.org/cgi/viewcontent.cgi?article=1258&context=bise

^v Gartner definition: gartner.com/it-glossary/information-platform-as-a-service-ipaas/

^{vi} Introduction: rickscloud.com/how-business-process-as-a-service-bpaas-works/

^{vii} Gartner definition: gartner.com/it-glossary/business-process-as-a-service-bpaas

^{viii} Definition of Business Process and Workflow: blog.goodelearning.com/bpmn/business-process-vs-workflow/

troubleshooting business processes. When this progress is not visible, it is impossible for the humans involved in the business processes to interact with the system in an orderly fashion.

- **Ability to include humans in the business processes.**

The important added value of a BPaaS solution over a workflow automation tool, is the ability to add human interaction to the processes. Many processes within a company, organization, school or university require human approval or decisions. Any system implementing BPaaS must allow human interaction in the business processes. If the system does not allow actions performed by humans to be part of the business process, the system is a workflow execution system and not BPaaS.

- **Low cost of entry, online, and pay-per-use.**

BPaaS solutions must be hosted online, have a low entry fee (if any), and, adhere to the cloud pay-per-use pricing model. This is a large differentiating factor, setting apart specific BPaaS solutions from the more general customer administration systems.

- **Cost-effective use of human resources and machine resources.**

As BPaaS solutions are cloud-based, they must make effective use of the scarce human and compute resources. This requirement ensures the pay-per-use pricing model is feasible. When the system does not make effective use of the resources, the system will become too expensive to be of any use to businesses.

3.2. Case Studies

A library of case studies is important for defining BPaaS, as it shows practical uses of business processes and which functional requirements need to be fulfilled. It also shows how business processes are executed, and why monitoring is important. Case studies are difficult to find, as most vendors only supply case studies at a sales pitch. The case studies that can be found, are often vague and do not include a full flow diagram of the steps executed. We will design a library of case studies including five different examples to show practical use of business processes for educational institutions, small businesses, large businesses and NGOs. Together these case studies give a good overview of practical use of business processes, including full flow diagrams which can be implemented using BPaaS.

The first case study describes the enrollment of students at universities or schools. This happens every year and contains a choice along the way. Some students are known from an earlier enrollment and already own a campus card; some students are new and still need to create their account. The end result should be that all students enrolled the following year have their badge and are registered in the university or school administrative system. This case study is an example where the scalability of cloud resources can reduce costs while still offering a fast response time. The enrollment only takes place once a year and has a large peak usage just before the deadline (students will be students).

The second case study is one every online retailer has to deal with: customers buying products. This case study describes, using logical components, the steps for the customer and selling company. The scalability for this second case study is not in regard to time, but in regard to size of the shop. Whether the online shop has only a handful of products, or millions of products, the result and response time will be the same. This is also known as *weak scalability* which, in contrast to the naming, is quite useful for online retailers.

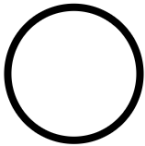
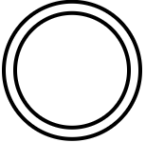


The third case study is probably well known to anyone who has ever had to get a payment reimbursed or approve of these reimbursement requests. The process behind these reimbursement requests is often fairly simple, but hard to implement. A perfect opportunity to automate using a BPaaS solution.

The fourth case study, starting a NGO, does not seem like a likely candidate at first. This study is provided to show that even processes one does not consider for automation at first, can be worth it on the long run.

The fifth case study, dealing with customer complaints, is the most complex of the studies. There are companies, and call-center employees, that are making a living off of this process. Especially with the rise of internet in the late 90's, the ability to complain, and the damage such complaints can do to the branding of companies, has risen tremendously. Every company should have a process in place for dealing with complaints.

These case studies are described in logical steps and depicted using the BPMN 2.0 notation [48]. From the BPMN 2.0 notation, we use the symbols listed in Table 3.1.

Table 3.1: BPMN 2.0 symbols used in Use Cases

Icon	Symbol name	Description
	Start symbol	Only appears once each Use Case. This is the starting trigger for the process. The actual trigger is also described using the label of this symbol.
	End symbol	Can appear more than once each Use Case. This is a final state for the process, no more processing occurs after this symbol is reached.
	State	In our Use Cases during this state either an action for the system or user is described, or an outcome of a previous choice which can be presented to the user. Optionally shows a user icon, to indicate it is a 'human in the loop' action.
	OR symbol	Indicates a choice. This symbol has two or more OUT-arrows, each arrow representing a process flow according to the outcome of the choice.

The visual models are created using a free online rendering toolkitⁱ. These models can be saved in an XML file format, and can be used as input for BPaaS implementations.ⁱⁱ

First Case Study: Student Enrollment

Chronological steps for this case study:

1. A student is required to register for a new year of education
2. If the student is known in the system, the student is required to log in
3. The student registers for an account
4. A photo is uploaded by the student for the account and badge
5. The student is notified by email when and where to pick up the campus card

Figure 3.1 shows this case study modelled in the BPMN 2.0 language.

ⁱ BPMN online rendering kit available at bpmn.io/

ⁱⁱ Currently, there is a conversion required from the online toolkit output, 1-on-1 input is one of the future work topics described in section 8.2 of Chapter 8.

Interesting features of this BPMN workflow is the interaction with the user at the OR symbol after state 1, which triggers the branch. Either the path 1->2->5->6 is taken, or 1->3->4->5->6.

All steps before the alternative paths come back together again are based on human input. After step 5, the system does not need any human input anymore and only notifies the student when and where to pick up the badge.

To determine when and where to pick up the badge, some calculation needs to be performed. This can be just a simple algorithm only dependent on the time of registration – or an elaborate algorithm taking into account the first lecture of the student or other personal information.

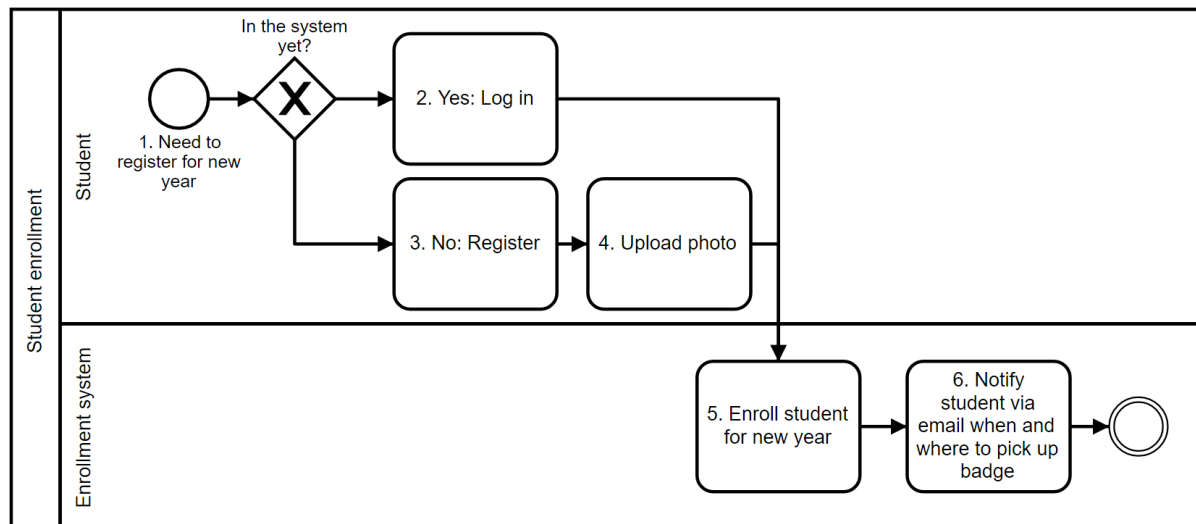


Figure 3.1: Student Enrollment case study.

For this case study, it is not important which algorithm is used for determining when and where to pick up the badge, rather the way it is called. During the design of the reference architecture, in the next Chapter, we will give a recommendation on how to implement such a computational question.

Second Case Study: Online Product Order

Chronological steps for this case study:

1. The customer visits the online shop
2. The customer decides which product to buy
3. Availability is checked
4. The customer needs to be logged in
5. A few shipping details need to be provided
6. The item needs to be payed
7. The product needs to be shipped to the customer

Figure 3.2 shows this case study, modelled in the BPMN 2.0 language.

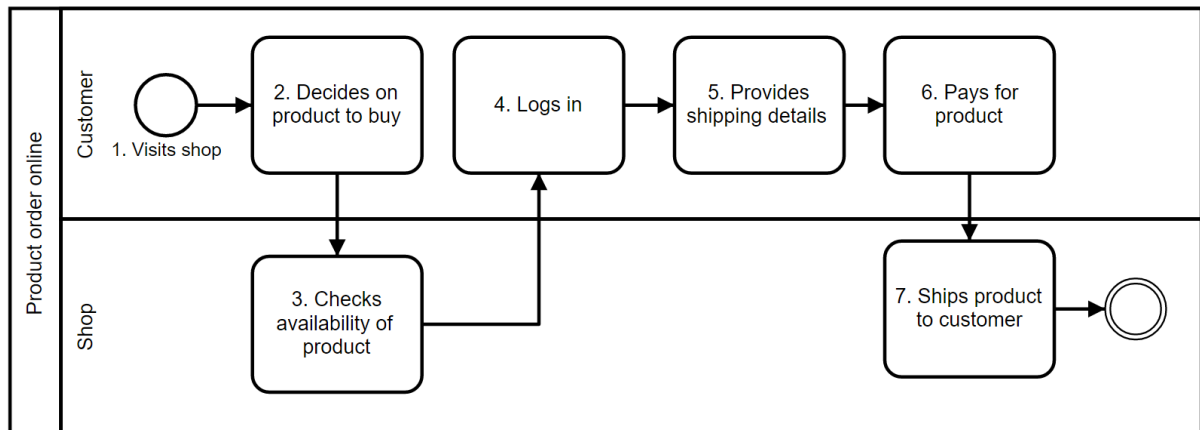


Figure 3.2: Online product order case study.

This is a slightly simplified look when compared to current offerings found at retailers online. Most online shops offer an ability to register for a notification when an item is out of stock, or show a “delayed shipping” warning and still allow the customer to order the product.

Here we assume the customer has an account at the store. The creation of a new account is already addressed in the first Case Study and has been left out of this Case Study for readability.

Third Case Study: Purchase Reimbursement Approval

Chronological steps for this case study:

1. Employee purchases items for work
2. Employee requests reimbursement for a certain amount
3. If amount is higher than 100 euro, proof of purchase is required
4. If amount is higher than 500 euro, reimbursement needs to be approved by manager
5. If amount is higher than 5.000 euro, reimbursement needs to be approved by CFO

Figure 3.3 shows this case study, modelled in the BPMN 2.0 language.

Fourth Case Study: Starting a new NGO

Chronological steps for this case study:

1. Define the NGO vision and write the mission plan
2. Search for NGOs with similar mission statements
3. Apply for funding
4. Spend funding
5. Report funding

Figure 3.4 shows this case study, modelled in the BPMN 2.0 language.

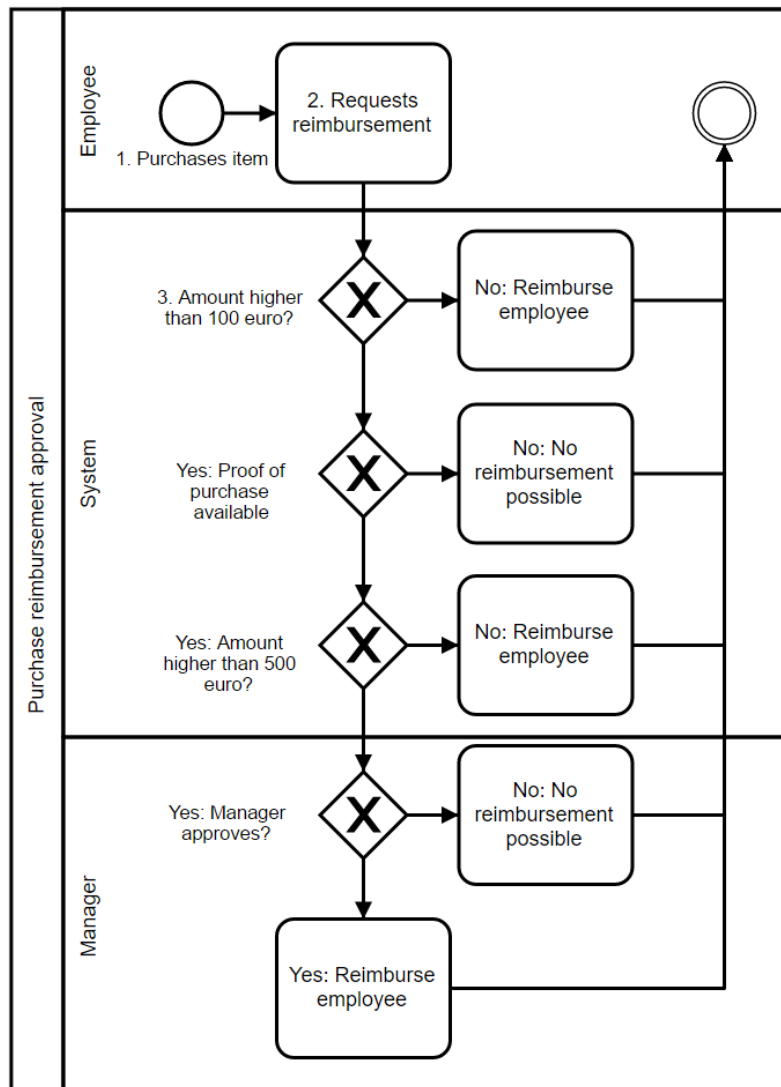


Figure 3.3: Purchase reimbursement approval case study.

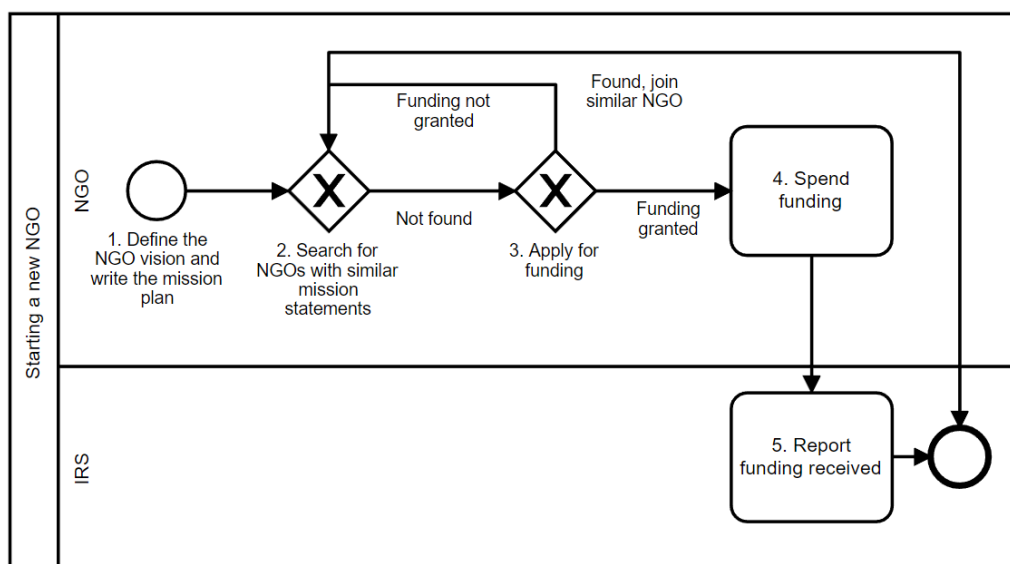


Figure 3.4: Starting NGO case study.

Fifth Case Study: Customer Complaint Resolution

Chronological steps for this case study:

1. Customer files a complaint via email
2. Complaint is routed to priority inbox if customer has a high value
3. Complaint is routed to relevant inbox otherwise
4. Employee reviews complaint and chooses compensation
5. Compensation is sent to customer and the customer is notified of this via email
6. The customer receives the email

Figure 3.5 shows this case study, modelled in the BPMN 2.0 language.

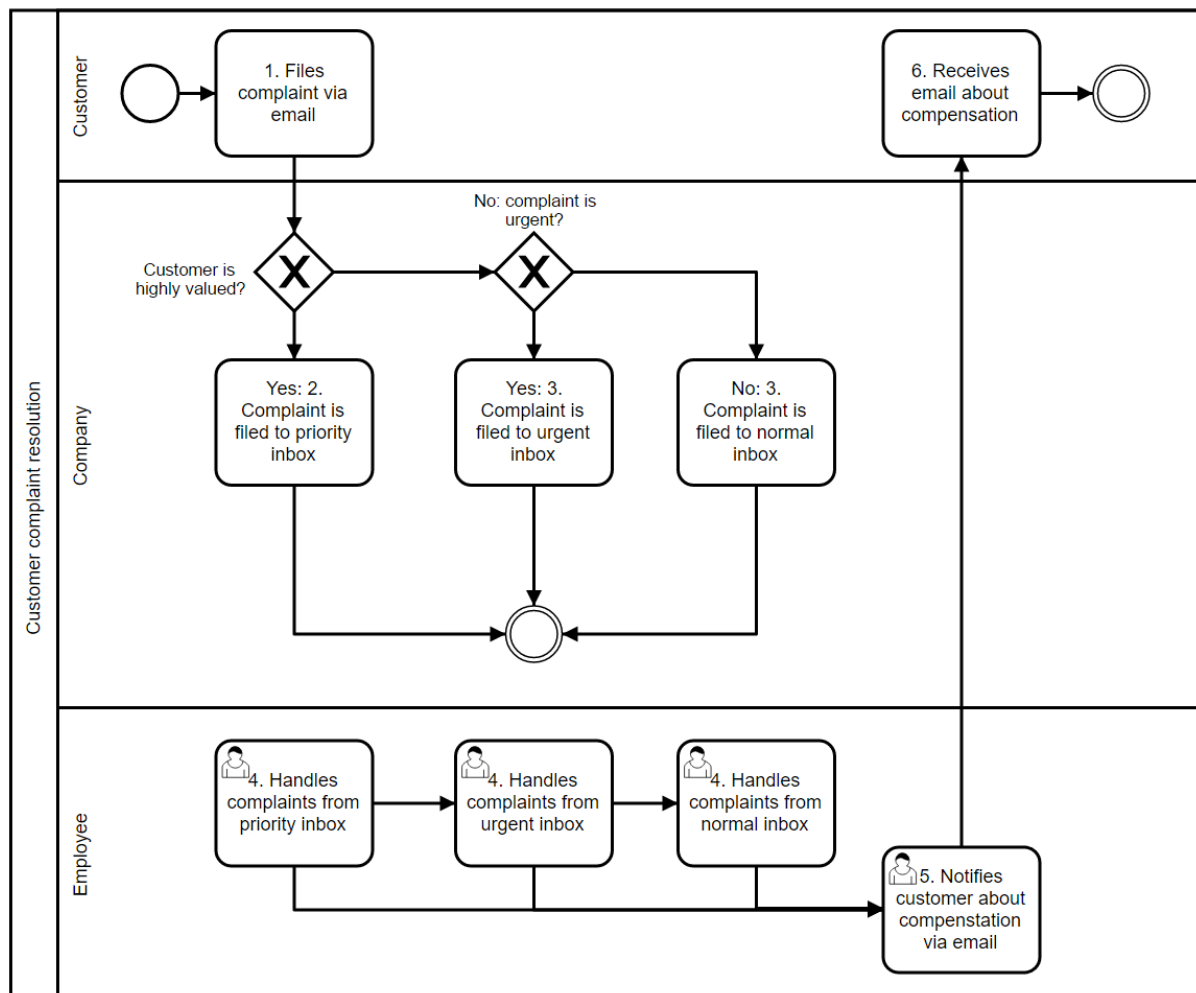


Figure 3.5: Customer complaint resolution case study.

3.3.Comparable definitions

We compare our definition of BPaaS, given in section 3.1, with other formal definitions of BPaaS found in computer science literature and leading online vendors. As stated in [49], research on BPaaS is relatively new, with only two relevant papers published before 2011. The same numbers show that there are only a few peer-reviewed papers mentioning BPaaS published each year. Commercial research institutes, like Gartner, are also relatively quiet on the topic of BPaaS. There is an index for BPaaS in the Gartner glossaryⁱ, but only a small selection of relevant articles on the topic of BPaaS can be found on their portal. IBM has been a leading software partner regarding business processes, the IBM definition of BPaaS, as listed below, is also taken into account.

We consider the following four definitions for comparison:

1. **Our definition from section 3.1:** *Any BPaaS solution, is a solution that offers the ability to visualize, execute, interact with business processes while cost-effectively making use of human and machine resources. The pricing model is a form of pay-per-use.*
2. **Gartner definition:** *The delivery of business process outsourcing (BPO) services that are sourced from the cloud and constructed for multitenancy. Services are often automated, and where human process actors are required, there is no overtly dedicated labor pool per client. The pricing models are consumption-based or subscription-based commercial terms. As a cloud service, the BPaaS model is accessed via Internet-based technologies.*
3. **Accorsi (2011) definition:** *BPaaS is a special SaaS provision model in which enterprise cloud offerors provide methods for the modelling, utilization, customization, and (distributed) execution of business processes. [50]*
4. **IBM definition:** *Business process services are any business process (horizontal or vertical) delivered through the Cloud service model (Multi-tenant, self-service provisioning, elastic scaling and usage metering or pricing) via the Internet with access via Web centric interfaces and exploiting Web-oriented cloud architecture. [51]*

Table 3.2: Comparison of BPaaS definitions.

Criteria \ Definitions	Definition 1	Definition 2	Definition 3	Definition 4
Modeling of business processes (visualization, customization, etc.)	Yes	No	Yes	No
Execution of business processes	Yes	Yes	Yes	Indirect ⁱⁱ
Considering Humans in the loop	Yes	Yes	No	No
Multi-tenancy explicitly mentioned	No	Yes	No	Yes
Cost-effectiveness mentioned	Yes	No	No	Indirect ⁱⁱⁱ
Pricing method defined	Yes	Yes	No	No

Table 3.2 shows the comparison of the four definitions taken into account. Our definition (1) and the definition given by Gartner (2) seem comparable. Our definition adds the modeling of business processes and the cost-effectiveness as requirements, but does not mention multi-tenancy. The definition given by Accorsi is limited, it considers BPaaS as a subset of SaaS. The definition given by IBM leaves out the modeling requirement, pricing method, and humans in the loop, and is in other parts too indirect for defining a reference architecture.

ⁱ BPaaS entry in the Gartner glossary: gartner.com/it-glossary/business-process-as-a-service-bpaas

ⁱⁱ “Delivered” could be taken as executed

ⁱⁱⁱ As the definition mentions elastic-scaling, this is indirectly considered a reference to cost-effective scaling

3.4. Summary

In this Chapter, we first introduced the formal definition and requirements for BPaaS, which is the conceptual contribution of this Chapter. We also contributed to the library of case studies for BPaaS with the introduction of 5 case studies. The functional and non-functional requirements, based on our definition and the presented case studies, are given in Chapter 4: Elastic-BPM: A Reference Architecture for BPaaS.

Finally, we compared our BPaaS definition with other BPaaS definitions found in relevant work, specifically with the Gartner, Accorsi, and IBM definitions. We conclude that our definition, introduced in section 3.1, compares favorable against all three of these definitions. While it does not explicitly mention multi-tenancy, this is also not excluded.

4. Elastic-BPM: A Reference Architecture for BPaaS

In Chapter 3, we have described a number of cases studies and given a clear definition of BPaaS. In this chapter, we will define a set of requirements based on these case studies and the BPaaS definition, and we introduce Elastic-BPM, a reference architecture for BPaaS.

The requirements listed in section 4.1 and Elastic-BPM, the reference architecture design described in section 4.2, are distilled from these use cases to define BPaaS more precisely. In section 4.6, we compare the architecture presented with other reference architectures.

Elastic-BPM is a conceptual contribution. It is a first reference architecture for business processes as a service (BPaaS) which takes the ‘*human in the loop*’ into account in the scheduling process. This gives new opportunities to optimize business processes, especially when combined with on-demand provisioning of infrastructure.

4.1. Requirements for a BPaaS architecture

In the following, we list the functional and non-functional requirements for a BPaaS solution, after which we shall explain why these requirements are sufficient.

Functional Requirements

- 1. Enable the business analyst to define processes consisting of common BPMN steps.ⁱ**
This requirement is necessary for the ability to adapt and change the business processes according to a changing business. Each business needs to change to stay current, without this requirement, any BPaaS solution quickly becomes obsolete.
- 2. Enable the business analyst to run processes on specific incoming data.ⁱⁱ**
Any business process needs data to function correctly, which makes this requirement necessary for the reference BPaaS architecture.
- 3. Enable the business analyst to inspect the status of running and the outcome of past processes.**
This requirement is necessary for businesses to improve. Being able to inspect the status of current and past processes is a crucial ingredient for any improvement.
- 4. Enable participants of processes to interact with the processes when required.**
The key difference between workflows and business processes, is the ability to interact with the process. Any BPaaS solution must support the ability to interaction with the processes.
- 5. Enable system administrators to monitor system health and response times.**
This requirement is necessary to verify that the BPaaS system is still functioning correctly. Just like the business analyst needs to be able to inspect the business status and outcome of processes, the system administrator needs to be able to verify the technical status of the BPaaS system.
- 6. Enable other systems to interact with the BPaaS solution using a comprehensive APIⁱⁱⁱ.**
This is a necessary requirement for BPaaS solutions, as BPaaS solutions are required to work together with other business applications. This interaction makes complex orchestration according to rules defined in business processes possible, enabling a higher level of automation than when the BPaaS system is only running in isolation.

ⁱ As defined in the previous Section

ⁱⁱ This data can be anything as long as it is partitioned in a JSON message

ⁱⁱⁱ The API should allow other systems get information on running processes and start and stop them

Non-functional requirements

a. Make use of current authorization and authentication standards.ⁱ

In order to allow business analysts, system administrators and especially other applications to interact with the BPaaS system, it is a necessity to adhere to the current authorization and authentication standards. Failing to do so will hamper all of the functional requirements.

b. Not have a single point of failure.

When BPaaS is used as central orchestration for business processes within a business, it is essential for business continuity to have this system running in a high-availability mode. Therefore, no part of the system should be a single point of failure.

c. Maintain a consistent state of the running and finished processes at all times.

For each functional requirement of a BPaaS system, a consistent state of running processes is a necessary non-functional requirement. For functional requirements 3 and 5, a consistent state of all finished processes is also required.

d. Scale according to demand.

Scaling of the BPaaS system is a necessary requirement for keeping the offering affordable for all businesses. Failing to scale will make it impossible to offer the system *as a Service*.

By implementing the functional requirements, any system implementing the Elastic-BPM BPaaS architecture will be able to: run processes defined by the business analysts and inspect them during and after they are run (REQ 1, 2, 3), let business users interact with these processes (REQ 4), let system administrators monitor the system and keep it up and running (REQ 5), and, let other systems run by the business interact with this BPaaS system (REQ 6).

As we are defining a reference architecture for a service, the non-functional requirements define the ‘*as a Service*’ requirements of authorization & authentication (REQ a), reliability (REQ b, c), and the ability to scale on demand (REQ d). REQ d we put to the test in Chapter 7 with a large set of experiments. When a reference architecture implements the functional and non-functional requirements, it sufficiently adheres to the BPaaS definition given in Chapter 3.

Following is a list of things *not* covered by the BPaaS architecture. Although this list is non-exhaustive, with it we cover the most frequently made assumptions of ‘*as a Service*’ offerings.

Requirements not covered

The BPaaS solution shall *not*:

i. Store and facilitate the management of customer data.

The storage and management of customer data has vastly different requirements from the storage and management of business processes. This data needs to be stored separately.

ii. Provide a means for outbound communication, with the exception of calling API endpoints.

Outbound communication is necessary for businesses of any kind, however, just like the storage and management of customer data, this has vastly different requirements than the storage and management of business processes and is thus not provided by BPaaS.

ⁱ Examples are: OAuth and OpenID

4.2. Reference Architecture Design

Elastic-BPM, the BPaaS reference architecture (shown in Figure 4.1, on the next page), is designed around the requirements from the previous section. This architecture is detailed enough to ensure all requirements can be met and yet general enough to enable implementations with all major cloud providers and technical components. This architecture is based on the Microservices architectural principles and thus consists of separate components, loosely coupled.

We define 3 main sections of the architecture: a) the functional section (explained in detail in 4.3), b) the scaling section (explained in detail in 4.4), and, c) the infrastructure section (explained in detail in 4.5). When following this reference architecture in implementing a BPaaS system, there is a solid fundament for fulfilling the requirements defined earlier in this Chapter.

The 3 actors defined in this reference architecture are:

- i. **The system admin.**
The system admin monitors machines, processes, and workers using both the Scaling API (component 5) and the BPaaS API (component 1). It does not interact with processes directly.
- ii. **The business analyst.**
The business analyst can inspect, define and run processes. The business analyst only has access to the BPaaS component.
- iii. **The participant.**
The participant is only able to interact with running processes and, just like the business analyst, only has access to the BPaaS component.

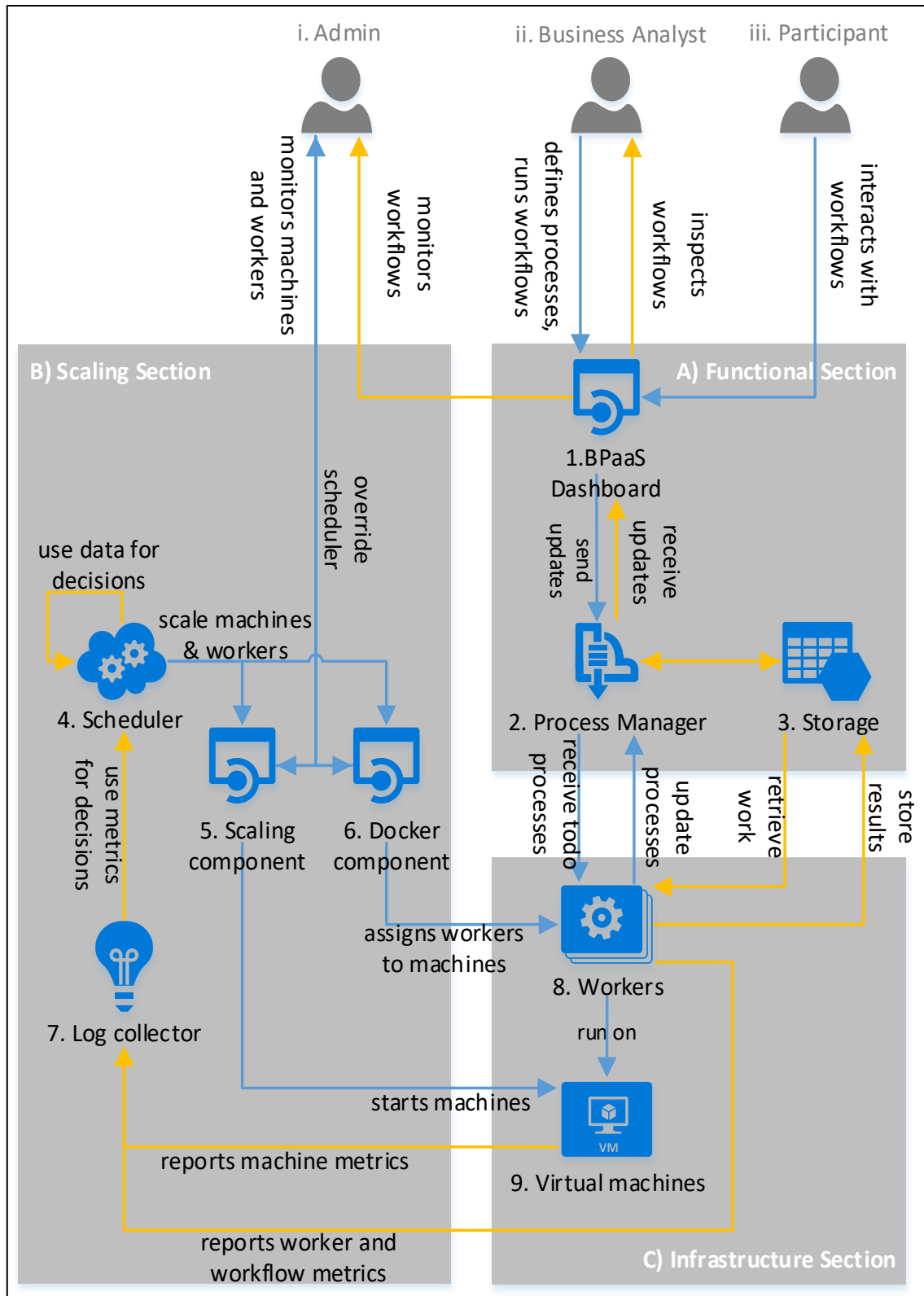


Figure 4.1: BPaaS Reference Architecture: blue arrows are control flows; yellow arrows are data flows.

4.3.Functional Components

The Functional Section, Section A in Figure 4.1, contains components regarding the workflow functionality, and interaction with the business analysts, system admins, and, participants. This is the functional core of the BPaaS system. In this section, functional requirements 1 through 4, and 6 are met. One of the components, component 1, also implements non-functional requirement a.

The three components in this section are:

1. The BPaaS Dashboard component.

A portal through which the business analysts (actor ii) and system administrator (actor i) can create, start, and, monitor business processes. It also allows the participants (actor iii) to interact with the processes. Technically an API is all that is required to meet these needs, however, a graphical user interface is often desired by the users. This component is responsible for the authentication and authorization of the users (non-functional requirement a).

2. The Process Manager component.

This component is the central hub in the Functional Section of the reference architecture. It makes sure the relevant data for processes is stored in component 3, the Storage component, provides workers with new work, and, updates the processes when the results from workers comes in. This component is responsible for keeping the processes in a consistent state, and receiving requests and updates regarding this state by all active worker components. It is important to implement this component efficiently, or it might become a bottleneck for the performance of the BPaaS system.

3. The Storage component.

A reliable and fast storage component for processes is required to meet functional requirements 3 (inspection of current and past processes) and 5 (inspection of health of the system). We recommend to use a database for this component – either a traditional structured database or a NoSQL database should suffice.

The interaction with all the actors is, in this action, restricted to component 1: the dashboard. For the business analyst (actor ii) and participant (actor iii), this is also the only interaction with the BPaaS system. The system administrator (actor i) interacts directly with other components, and then only in the Scaling Section.

The Scaling Section, shown in Figure 4.1 as section B, contains components regarding the scaling of the infrastructure, the scheduling functionality, and, provides interaction with the system admins. This section specifically fulfills non-functional **requirement d: scale according to demand**, which is essential for any ‘as a Service’ offering. These components, specifically the Scheduler component, contain the innovative algorithms for scheduling and scaling, which we will validate extensively in Chapter 7.

The Infrastructure Section, Section C in Figure 4.1, contains components on which the workflows are executed. This section makes sure the Functional Section, Section A, in which the workflows are created, monitored, and interacted with, can fulfill the requirements set out in section 4.1 of this Chapter. The Infrastructure Section depends on the Scaling Section, Section B, for provisioning and allocation of its components and communicates with the Process Manager component from Section A to receive and update information on running processes.

Next, we describe the scaling and infrastructure components in more detail.

4.4. Scaling Components

The components in the scaling section are:

4. The Scheduler component.

The Scheduler is the component which contains the main logic for this section, it acts on a set interval on information gathered from component 6, the Log Collector. This information is used as input for the selected provisioning policy, which can result in scaling actions sent to component 5, the Scaling component, and to component 6, the Docker component. The provisioning policies used by this scheduling component are introduced in Chapter 5.

5. The Scaling component.

The Scaling component executes the actions received from the scheduler component, these actions can result in changes in the Infrastructure Section: The Virtual Machines, which we will explain in more detail in the following section in this Chapter. The scaling component starts and shuts down virtual machines by calling functions provided by the cloud platform.

6. The Docker component.

The Docker component, like the scaling component, executes actions issued by the Scheduler component. These actions can result in changes in the Infrastructure Section, for the worker components, as these are run using a Docker service. The Docker component can set the amount of worker components per node by calling the Docker API and creating or updating the *service* which the workers are part of.

7. The Log Collector component.

The log component, which collects execution logs and system metrics, is often left out of reference architectures. However, as we are actively using this data to improve the performance of the system, this is a crucial part of the reference architecture for BPaaS. The implementation of this component requires storage and retrieval of non-structured data in real-time. Any other component should be able to write logs to this component (not shown explicitly in Figure 4.1), the Docker containers and Virtual Machines also report metrics to this component (shown in Figure 4.1).

The Scaling and Docker components are also accessible by the system admin (actor i), to enter manual commands for scaling the Infrastructure Section, or to override actions given by the Scheduler component. A good example of the system admin overriding the actions from the Scheduler component, is to give specific workflows priority over others, or to manually increase the capacity of the system when a large batch of workflows is about to enter the system.

Apart from the override commands from the system admin and logs being written to the log collector component, the scaling section only has connections with the infrastructure section. The interaction is executed by the scaling component (component 5) or the docker component (component 6), giving instructions to the worker components (component 8) and the virtual machine component (component 9) respectively. The Infrastructure section containing these components is explained in more detail in the following section of this Chapter.

4.5. Infrastructure Components

The two components within the infrastructure section are:

8. The Worker component.

The Worker component retrieves information on processes from component 2, the Process Manager. The results and inquiries from the process executed are returned to the Process Manager component after the work has been executed. The Worker component can and should be instantiated multiple times to fulfill non-functional **requirement d: scale according to demand**.

9. The Virtual Machine component.

All instances of the Worker component run on the Virtual Machine component. There can be 0 or more Virtual Machines live at any moment in the lifetime of the system, these are started and shut down by the Scaling component, component 5 of the Scaling Section.

Both the Worker and the Virtual Machine component report metrics to component 7, the Log Collector.

Interaction with any of the actors is indirect: the system admin actor (actor i) can influence the scheduler component, to start or shut down virtual machines. The business analyst (actor ii) and the participant (actor iii) interact with the running processes, which in turn run on one or several worker components.

4.6. Comparable Architectures

The unique aspect of the reference architecture shown in Figure 4.1, and BPaaS in general, is the participant actor (iii). The participant, using the BPaaS Dashboard component, directly influences running workflows. Most aspects of business process workflows revolve entirely around interaction with this actor, causing them to pause until the '*human in the loop*' has made a decision or entered some required information. This interaction influences the Scheduler component and its provisioning policies and allocation algorithms directly, as humans are an unpredictable and unreliable variable.

Using log data combined with metrics, the reference architecture allows for new provisioning algorithms where the '*human in the loop*' is taken into account. This combination is missing in architectures for workflow execution, for example as found in [52]. This architecture has comparable components and sections to the BPaaS reference architecture, but cannot fulfill functional requirement 4: **enable participants of processes to interact with the processes when required**, which makes it unfit for a BPaaS solution.

The architecture of Semantic Business Process Management Systems (SBPMS), described in [53], is published in 1998 is created specifically for the automation of business processes. There are a few similarities with our reference architecture. Specifically, the storage of historic processes for data mining is shared. The SBPMS architecture, however, does not focus on scheduling of processes, but rather the basic execution of them, and the discovery of interesting properties of the process. There is also no consideration of the '*human in the loop*'.

Apache Hadoop is a new standard in the workflow execution and data science software space, it is an open source framework for distributed storage and processing. The architecture for Hadoop, as presented in [54], can be combined with the presented reference architecture. The worker component and the storage component can be fulfilled by using a Hadoop implementation. This is beneficial only when most of the compute tasks in the workload are data-heavy and the scale of work is known in advance, due to the way the Hadoop architecture scales.

Apache Spark, which seems to be quite a bit more efficient than Hadoop in several big data use cases as presented in [55], can be used in the same manner as Hadoop: replacing the worker and storage

components with a Spark implementation. Scaling spark on a container cluster is also a possibilityⁱ, which makes it better suitable for BPaaS than Hadoop, in combination with the scaling algorithms presented in this thesis.

The SAP HANA database architecture, as described in [56], has a different approach to executing workflows and procedural logic. It is a single monolithic system, scaled over one or more large compute units, taking advantage of all available memory and CPU power to execute analytical and transactional workloads. While this approach is a great contender for the traditional database systems and newer column-stores, it does not scale according to demand and does not accommodate the ‘human in the loop’ business processes requirements as outlined in section 4.1.

4.7. Summary

In this Chapter, we first introduced the requirements for BPaaS. The functional and non-functional requirements presented, are based on the case studies and definition presented in Chapter 3: BPaaS Definition.

We also introduced the reference architecture for BPaaS, which is the conceptual contribution of this Chapter. It is the first reference architecture to take the ‘*human in the loop*’ into account when scheduling business processes.

Finally, we compared our reference architecture with reference architectures found in relevant works. Comparisons are made with a recent architecture for the Elastic Cloud Platform, an older architecture for a Semantic Business Process Management Systems platform, recent developments in the workflow execution and data science space (Apache Hadoop and Apache Spark), and, SAP HANA: a database capable of analytical and transactional workflows.

ⁱ Scaling Spark on a Kubernetes cluster: blog.madhukaraphatak.com/scaling-spark-with-kubernetes-part-7/

5. Scheduling in Elastic-BPM

The formal definition for BPaaS, given in section 3.1, includes the cost-effective use of human and machine resources. Achieving this cost-effectiveness requires the scheduling of flows and provisioning of resources. These topics are explained in more detail in the following sections.

The first section in this chapter will define the metrics and explain our optimization. The second section of this chapter will dive into the different aspects of provisioning and allocation in the BPaaS system, using virtual machines and Docker services.

Our conceptual contributions are the new *learning* provisioning policies, presented in section 5.2.

5.1. Metrics

We make the distinction between system- and user-oriented evaluation metrics, as introduced in [52]. System-oriented evaluation metrics quantify the different aspects of provisioning: over-provisioning, and under-provisioning. These metrics do not take into account the performance of the business process execution, but do amount for the difference in cost occurred by using the BPaaS system. They also give us an insight in the upper limit of business process performance.

We introduce system-oriented evaluation metrics, and system-utilization metrics in this Chapter. We will introduce user-oriented and cost-metrics in Chapter 7, when we validate the BPaaS implementation.

System-oriented evaluation metrics

The system-oriented evaluation metrics we define are:

Demand: the minimal amount of resources required for fulfilling an objective. In this context, as we deal with workflows, the *demand* is the number of workflows that are ready to execute.

Supply: Total number of provisioned resources. In our case, this is the number of active nodes in the system – idle and busy.

Under-provisioning accuracy: The average fraction by which the demand exceeds the supply. The under-provisioning accuracy, a_U , is defined as follows:

$$a_U := \frac{1}{T \cdot R} \sum_{t=1}^T |d_t - s_t| \cdot \Delta t,$$

where d_t is the demand and s_t is the supply, T denotes the time horizon of the experiment, R denotes the available resources, $|x|$ is the absolute value of x , and Δt is the time elapsed between two measurements.

Over-provisioning accuracy: The average fraction by which the supply exceeds the demand. The over-provisioning accuracy, a_O , is defined as follows:

$$a_O := \frac{1}{T \cdot R} \sum_{t=1}^T |s_t - d_t| \cdot \Delta t,$$

where we take a look at the difference between the supply, s_t , and the demand, d_t .

We choose not to take into account the *wrong-provisioning timeshare*, and *instability* metrics, to limit the scope of this thesis. This can be a topic for future research, if results indicate there are potential gains in that area.

System-utilization metrics

The system-utilization metrics we define, per machine, are:

CPU load: as reported by the Linux operating system^{en.wikipedia.org/wiki/Load_(computing)}

Memory usage: number of GBs of RAM in use

Network usage: number of GBs received by the network interface

These metrics have been defined before, for example in [57] and [58]. While the metrics seem simple enough, it is tough comparing apples with oranges, which can be the case if there are different types of processors, disks and memory configurations in the used machines. For our testing purposes, we have been using the same type of machine with identical CPU, memory and network capacity.

The machine used for the experiments is described in detail in Chapter 7, where the setup of the environment is explained.

5.2. Processes in BPaaS

Processes in BPaaS are a directed graph of tasks, to allow the users to model their use cases as accurately as possible. 5 examples of use cases which can be modelled according to these specifications have been given in Chapter 3.

Each task in these processes has one of the following 5 types:

- CC is a CPU intensive task: computing prime numbers,
- CN is a network intensive task: downloading a file,
- CI is an IO-intensive task: writing random bytes to disk,
- HE is an easy task for humans, and,
- HH is a hard task for humans.

The tasks also have a size. For the computer-related tasks, this size is either S for Small, M for Medium, or L for Large. For the human tasks, the size is a number, which is the number of seconds the task is expected to take.

In the directed graph, work can only be started on a task if all the nodes from all incoming are marked completed, or if there are no incoming edges. This is a simple and effective check, making sure the set of tasks is finished in the desired order.

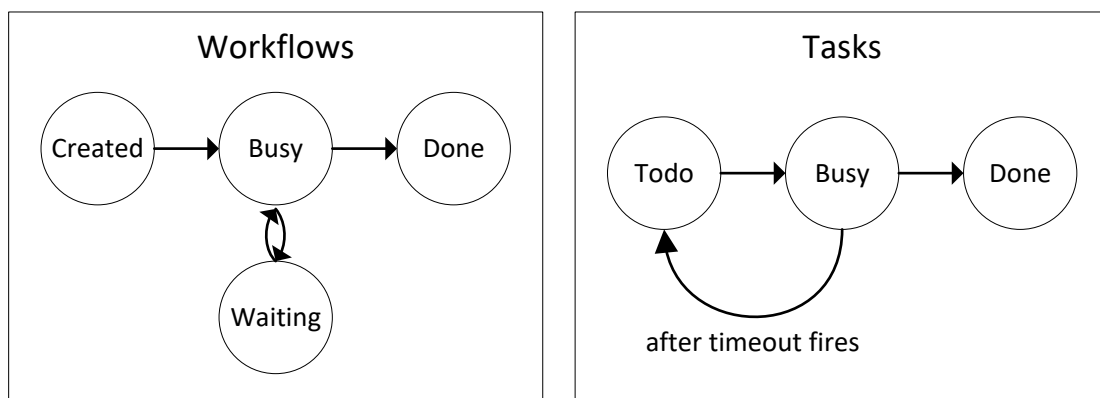


Figure 5.1: Status changes of workflows and tasks in Elastic-BPM.

When a process is first created, the status is set to 'Created'. When one or more of the tasks in a workflow are being worked on, the workflow status is changed to 'Busy'. When at least one task is finished, there are no tasks being worked on currently, and there are still tasks left unfinished, the workflow status is 'Waiting'.

When all tasks are finished, the workflow status is changed to 'Done'. Tasks themselves are either 'todo', 'busy' or 'done', when a task is 'busy' and a timeout occurs, the state is set back to 'todo'. The implementation of these processes and tasks is explained in more detail in Chapter 6.

5.3. Provisioning resources

For BPaaS solutions, the most important part of scheduling, is the provisioning of resources. As resources are billed per minute, making optimal use of these resources can have a large influence on the cost per workflow. Figure 5.2 shows the scheduler component (4), the component is depends on for information on the running system (7), and the components it directly influences (5 & 6). In this figure the yellow arrows are data-flows, the blue arrows are control-flows. The instructions for starting and stopping machines it gives to component 5, the scaling component, directly influence the running costs of the system.

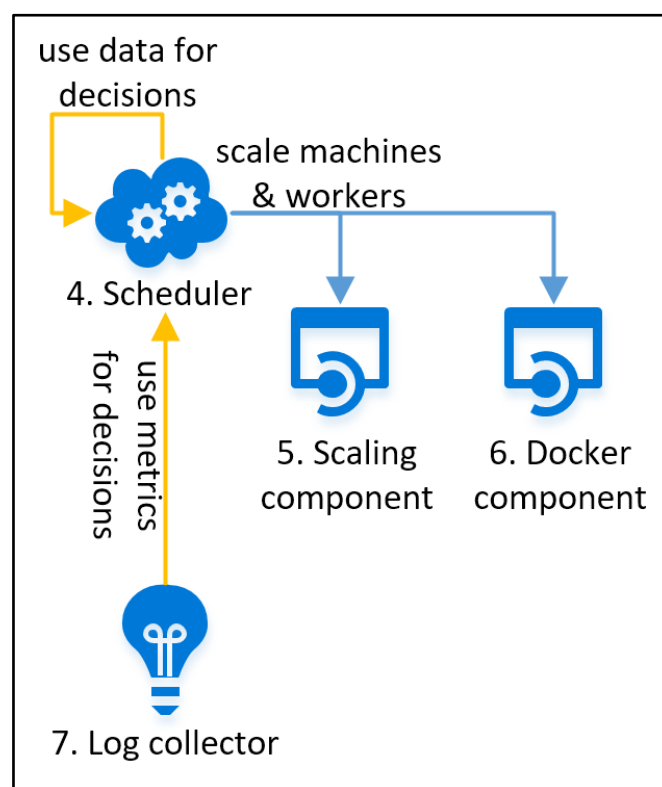


Figure 5.2: The scheduler component and the components it influences.
(Blue arrows are control flows; yellow arrows are data flows)

There are quite a few provisioning policies that can be implemented when scheduling workflows and provisioning resources. The most naïve is the *Static* provisioning policy, which starts all possible machines and does not scale down. This policy is a great baseline, as the costs are predictable and the performance of the system using a set number of machines can be benchmarked.

Another basic policy, the *OnDemand* policy, is a policy that takes into account the current load of the system. When the load of machines in the system is higher than a set threshold, a new machine is started. When the load of machines in the system is below a set threshold, the least busy machine can be shut down. This policy has been described in [9] and [59]. We have implemented this policy to compare this known scaling policy with the baseline Static policy and our new Learning policy.

Both the Static and OnDemand policies do not take into account the running business processes or heuristics of the system, and as such are “open loop” scheduling policies from [60] and [61].

Our contribution is a variation on the Feedback Control Real-time scheduling (FCS) algorithm: The *Learning* policy. The Learning policy takes into account the current running business processes and builds heuristics of these processes, to scale machines optimally. Table 5.1 gives an overview of these provisioning policies.

Provisioning Policy	Determines scaling on	Complexity	Gets better over time
Static	-	Simple	No
On Demand	Load of the system	Medium	No
Learning	Specific metrics	Complex	Yes

Table 5.1: Overview of provisioning policies.

In the following sections, we describe these provisioning policies in more detail.

5.3.1. Static provisioning policy

The Static policy provides a baseline metric: the performance is maximized, as is the cost. This policy does not shut down machines during the full run of the system.

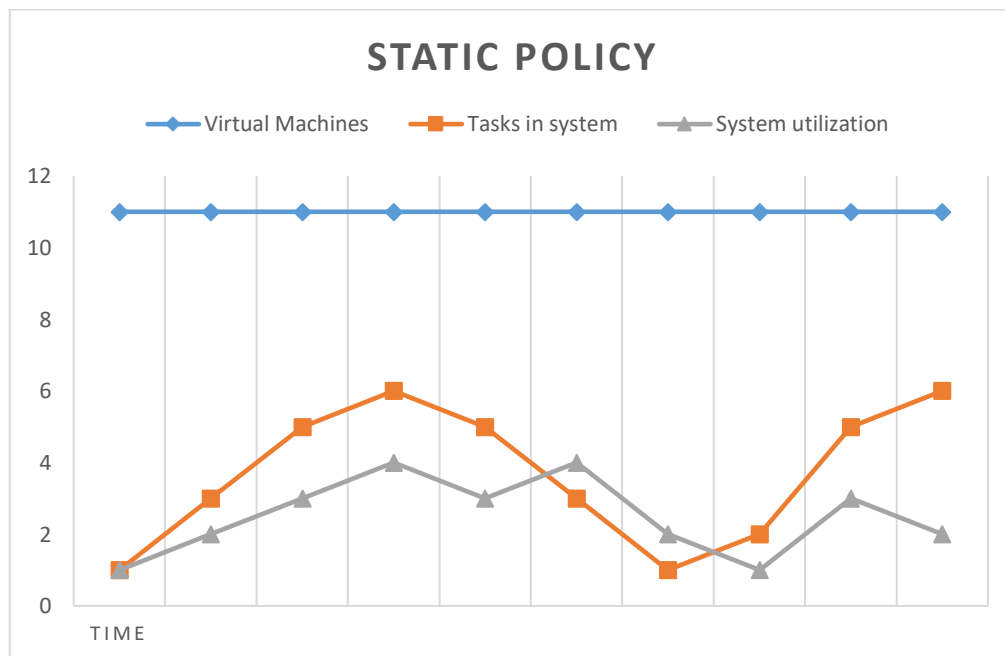


Figure 5.3: Static policy depicted over time for an example workflow.

Figure 5.3 shows an example of the scaling of virtual machines over time, the tasks in the system and the system utilization for an example workflow.

5.3.2. On Demand provisioning policy

When using the OnDemand provisioning policy, the system provisions a minimal set of machines (n), and then adds machines when high system load is detected.

Figure 5.4 shows an example of the scaling of virtual machines over time, the tasks in the system and the system utilization for an example workflow.

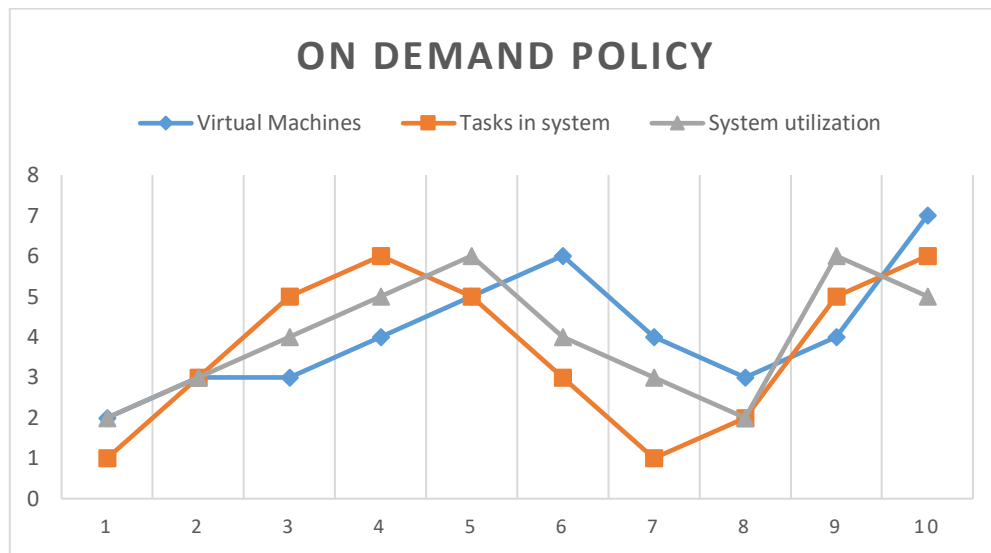


Figure 5.4: On Demand policy depicted over time for an example workflow.

5.3.3. Learning provisioning policy

The “Learning” provisioning policy is an extension on the “OnDemand” provisioning policy. A minimal set of machines (n) is provisioned at the system start, when the machines in this initial set register a high system load *and* there is incoming demand, a new machine is provisioned.

The incoming demand is based on heuristics and feedback the policy has collecting during runtime regarding the chance on *human interaction*. This demand is then inspected, and according to the demand, the new machine is either added to the volatile or, if needed, to the non-volatile pool.

We introduce three variants of this provisioning policy:

- Scaling based on number of tasks in the system
- Scaling based on heuristics on tasks in the system
- Scaling based on feedback gained on tasks in the system

The first variant, variant a, requires no information on the tasks themselves, only the number of tasks in the system. The second variant, variant b, requires basic heuristics on the tasks in the system: if the tasks are classified as S, M or L for non-human tasks. The last variant, variant c, starts out equal to variant b: using the S, M and L classifiers, but improves these estimations based on gathered data while running. We have implemented the first variant, variant a, in the Elastic-BPM system. Variant b and c are left for future work, as explained in more detail in section 8.2.

Figure 5.5 shows an example of the scaling of virtual machines over time, the tasks in the system and the system utilization for an example workflow.

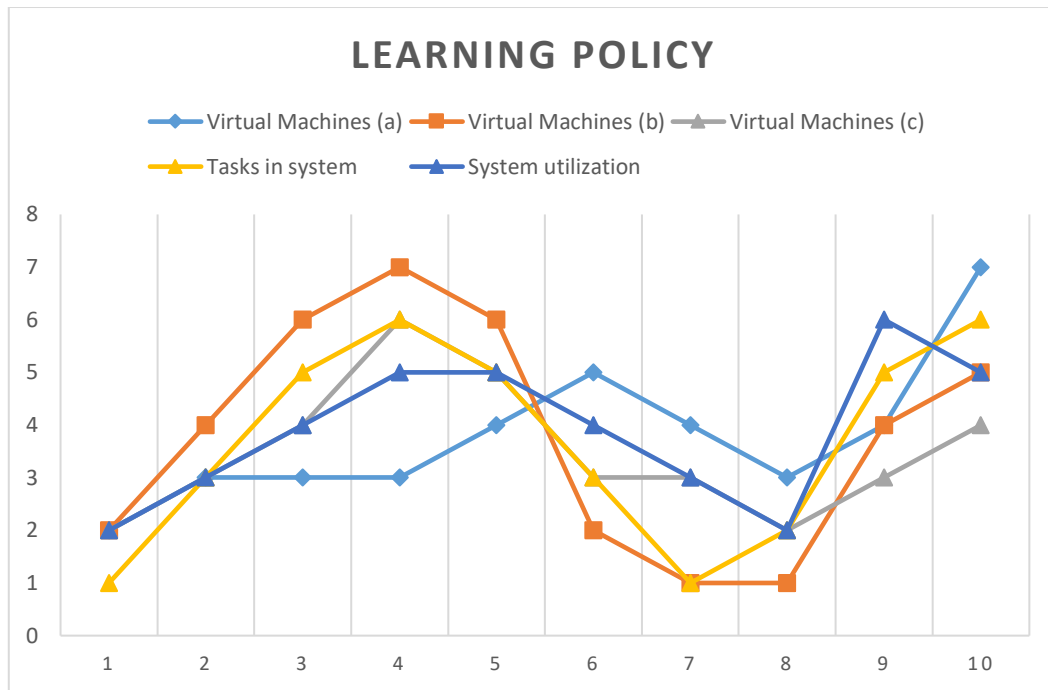


Figure 5.5: Feedback Learning policy depicted over time for an example workflow.

5.4. Summary

In this Chapter we introduce the demand, supply, under-provisioning and over-provisioning accuracy system-oriented evaluation metrics and the CPU load, and memory and network usage system-utilization metrics.

We also introduce 3 families of provisioning policies: Static, OnDemand, and Learning. In the Learning provisioning policy family, we introduce scaling based on number of tasks, scaling based on size of tasks, and scaling based on heuristics. This is also the conceptual contribution in this Chapter.

The Static, OnDemand and Learning with scaling based on number of tasks provisioning policies are implemented in the prototype implementation described in the next Chapter. The result of experiments performed on this this implementation, are presented in Chapter 8.

6. Implementation of the Elastic-BPM Prototype

In the previous Chapters we have defined BPaaS, the requirements for a BPaaS system and a reference architecture to implement BPaaS systems. This Chapter describes how we have implemented a prototype for BPaaS: Elastic-BPM.

We start this Chapter with a short introduction to Agile working in section 6.1, then we introduce twelve rules of thumb for creating a modern application in section 6.2. Section 6.3 introduces how Docker is used to create the different components for Elastic-BPM, for which we describe the implementation in section 6.4. We conclude this Chapter with a summary in section 6.5.

Our technical contribution this Chapter is the implementation of a prototype according to the BPaaS definition and reference architecture.

6.1. The Agile way of working

Every year more software is created than the year before itⁱ. It has been theorized by Robert C. Martin, that the number of people active in the IT business doubles every five yearsⁱⁱ. From this follows, that half of the people active in the IT business at every given moment has less than six years of experience. The large amount of software created each year combined with the large amount of inexperienced IT professionals leads to an ever-growing amount of hard to maintain software.

Code quality, an aspect of developing code that has been largely neglected for years, is now becoming increasingly important. When the quality of code is low, the difficulty of maintaining this code goes up. This leads to high costs of adapting and changing the application, both in time and money. With the current speed of change and rate of innovation in the software market, this is quickly becoming a large problem for companies and research institutes. To battle this problem, more and more companies are using the Agile methodologies for creating software. Agile is also steadily finding its way into the courses taught at universities.

Agile is a broad collection of practices and frameworks to use when developing software [62], [63]. Though it has been used in other areas of work, software development has been the main focus of the group of people that created the Agile manifesto in 2001[64]. This manifesto was written to underline the needs of the best software engineers in the field, and to bring about changeⁱⁱⁱ. The fundamental difference with other methodologies is the large emphasis on learning: continuous improvement is the norm. Do the minimal required planning up front to get started fast & go out and learn!

After an amount of development time on a software project, there is a tipping point after which the team spends more time on re-factoring or completely rewriting code than on writing new code. This is natural, especially when working in a changing and competitive environment. Plans change, requirements change, and thus, the code also needs to change. When the team grows and the lines of code, functions, and components grow as well, the cost of changing the software increases, which reduces the responsiveness to change. Figure 6.1 shows the cost of change and responsiveness over time, the difference between the actual cost of change and optimal cost of change is called technical debt^{iv}.

ⁱ Information on software created on Github: [github.info/](https://github.com)

ⁱⁱ Stated by Robert C. Martin on his blog: blog.cleancoder.com/uncle-bob/2014/06/20/MyLawn.html

ⁱⁱⁱ Information on the Agile manifesto: techbeacon.com/survey-agile-new-norm

^{iv} Background information on technical debt: mnapoli.fr/should-you-really-write-ugly-code-no/

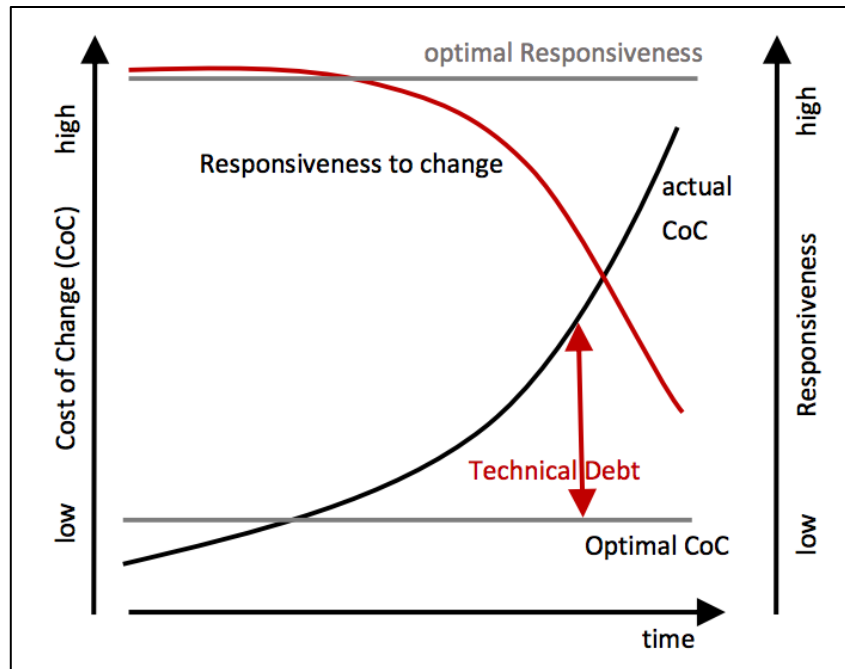


Figure 6.1: Visualization of technical debt.

There have been quite a few techniques proposed to battle technical debtⁱ, among them:

1. Test Driven Development (TDD)
2. Behavior Driven Development (BDD)
3. Domain Driven Design (DDD)
4. Acceptance Test Driven Development (ATDD)

TDD is the most used technique of these [65]–[67]. When using TDD to develop software, it is common to first write a failing test describing the purpose of the function to create or change. This test should then fail because either the function does not exist yet, or does not yet behave as required. The next step is to write the minimal required code to make the test succeed. This code should be as simple as possible. The next step is to run all the other tests in the project and make sure the function passes these tests as well, this is called testing for regression. If all other tests also succeed, it is time to improve the newly written code to make sure it fits the standards, while still succeeding the tests. Once this is done, it is time to add a new test, for the next piece of functionality to be added. Figure 6.2 shows this flow for TDD.

When developing, it is important to minimize the time between creating, testing, and deploying. In the best case, when you change something, it should be re-built and re-deployed automatically and fast. When creating a web-based UI, this is possible, as most web components either do not need to be compiled, or compilation is fast enough to see a change nearly instantly. For most other software, this is not yet possible.

ⁱ Different techniques to battle technical debt: assertselenium.com/atdd/difference-between-tdd-bdd-atdd/

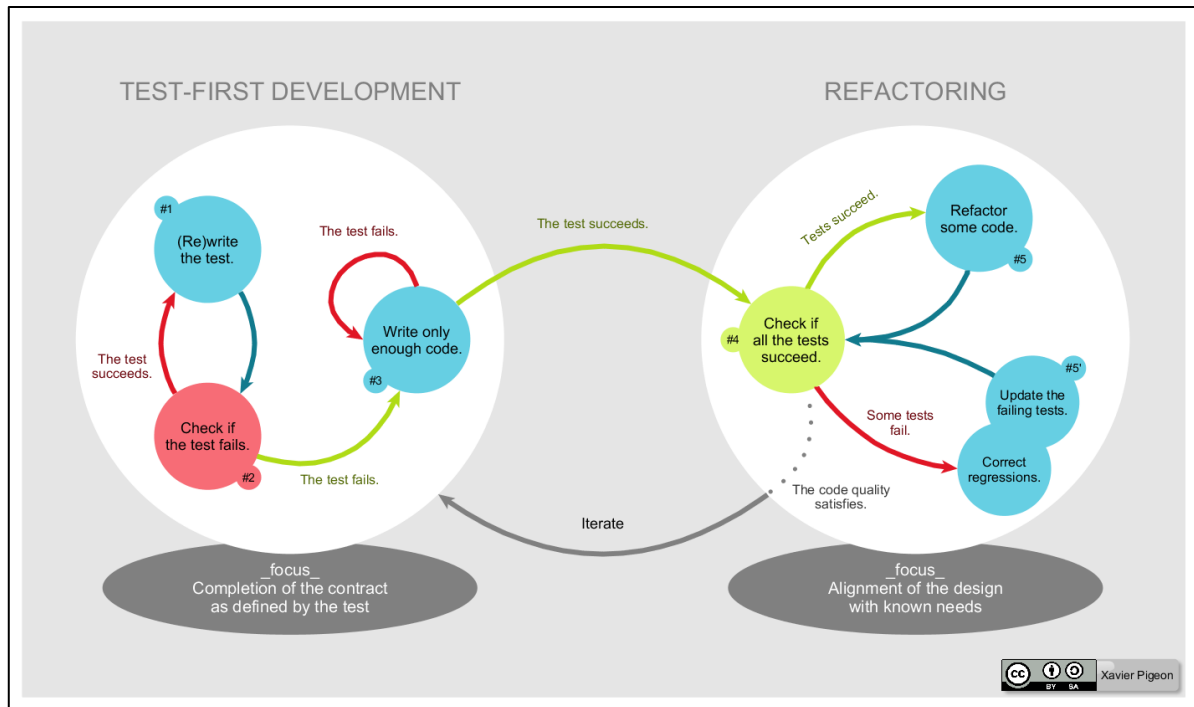


Figure 6.2: Test Driven Development (TDD) lifecycle.

To shorten the time between creating, testing, and deploying, it is worthwhile to invest time to plan and implement automated testing and deployment before starting development [68]. For Elastic-BPM, using Docker containers and the Docker Cloudⁱ made the testing and deploying a lot faster, and fully automated. This automation reduced distractions and human error. All in all, the creation and configuration of the automatic compilation and deployment took a few days, but has saved countless hours of manually starting/stopping components and debugging.

When implementing Elastic-BPM, elements of Agile working and TDD have been used.

6.2. The twelve-factor App Methodology

Running software inside multiple containers, and often on more than one machine, is quite different from the standard three-layered application (also known as traditional applications). Most current best practices and standards are geared towards traditional applications, and will most probably cause some serious problems when applied to software or applications running inside multiple containers on multiple machines, often referred to as Apps or SaaS apps – or more traditionally “Distributed Applications”.

The team behind Herokuⁱⁱ, one of the earliest cloud platforms, has published a methodology for building SaaS apps: the twelve-factor app methodology. Not every factor is still relevant in every project, the frameworks around Docker alleviate some of the constraints originally applied in the cloud platforms, but most of them are still relevant enough to mention this methodology here.

ⁱ Information on the Docker Cloud: cloud.docker.com

ⁱⁱ Heroku webpage: heroku.com

The twelve factors to consider when creating a modern application are:

1. **Codebase:** one codebase tracked in revision control, many deploys
2. **Dependencies:** explicitly declare and isolate dependencies
3. **Configuration:** store configuration in the environment
4. **Backing Services:** treat backing services as attached resources
5. **Build, release, run:** strictly separate build and run stages
6. **Processes:** execute the app as one or more stateless processes
7. **Port Binding:** export services via port binding
8. **Concurrency:** scale out via the process model
9. **Disposability:** maximize robustness with fast startup and graceful shutdown
10. **Development and Production parity:** keep development, staging and production as similar as possible
11. **Logging:** treat logs as event streams
12. **Admin processes:** run admin and management tasks as one-off processes

In the following sections and Chapters, we will refer to these twelve factors while describing the components we implement for Elastic-BPM.

6.3. Creating Microservices for Elastic-BPM

Docker is the current market leader when using Container technology. There is quite a large amount of Docker Containers available in the Docker Hubⁱ (soon to be migrated to the Docker Storeⁱⁱ). A large part of these Containers is only usable on a Docker system using a form of Linux as host OS. From version 2016 upwards, Windows Server also supports Docker and Docker Containers. Figure 6.3 shows a side by side comparison of the differences between a Windows Server and a Linux server running Docker.

Most programming languages run on both Linux and Windows, so the difference between Linux and Windows Containers is small. The most noticeable exception is the .NET software stack. The .NET software stack includes C#, VB, ASP.NET, and does not natively run on Linux-based operating systems or Containers. Microsoft is currently adapting parts of the .NET stack to also run on Linux-based operating systems: .NET Coreⁱⁱⁱ and ASP.NET Core^{iv} are examples of this. Microsoft has also open sourced these solutions: they can now be found at the Microsoft public Github repository^v.

As currently the most mature tooling regarding Docker Containers is only compatible with Linux-based Containers, Elastic-BPM is implemented using Linux-based Containers.

Abraham Maslow said in 1966: “I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail”^{vi}. Important takeaway here is:

- a) Get to know the relevant tools at least a bit before deciding on the best one, and,
- b) Decide which tool to use based on the requirements at hand.

The sheer number of tools available and the uncertainty in how they fit the requirements, make this no simple task when creating a microservices based solution. Often personal preference of the developer or team of developers is an important part in the choice of tools.

ⁱ Location of the Docker Hub: hub.docker.com/

ⁱⁱ Location of the Docker Store: store.docker.com/

ⁱⁱⁱ Documentation on .NET core on Ubuntu: microsoft.com/net/core#ubuntu

^{iv} Documentation on ASP.NET core: docs.microsoft.com/en-us/aspnet/core/

^v Public Microsoft Github repository for .NET: github.com/Microsoft/dotnet

^{vi} [77], page 15 (en.wiktionary.org/wiki/if_all_you_have_is_a_hammer_everything_looks_like_a_nail)

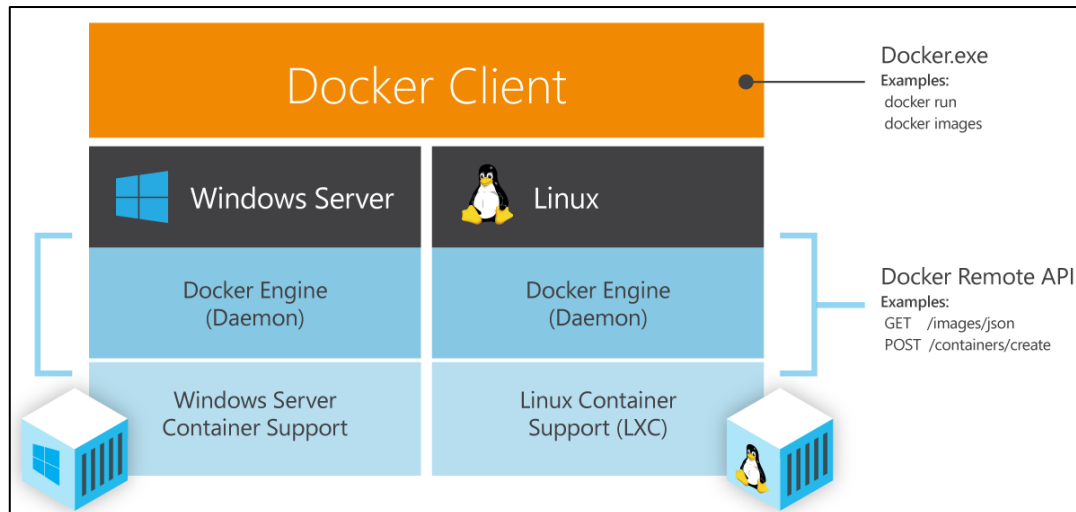


Figure 6.3: Windows and Linux support for Docker.

We have chosen to use Node.jsⁱ for the server components and HTML5^{w3.org/TR/html5/} (web interface) for the UI to implement Elastic-BPM.

Node.js for the server components, because:

- The support in Docker is great: there is even an official Docker imageⁱⁱ,
- Most of the functions within Node.js are event-driven, which fits our use-case,
- It is quick to set up a working REST API using expressⁱⁱⁱ,
- The client library for consuming REST APIs works and is well documented^{iv},
- The Socket.io^v library works out of the box for WebSockets,
- There is a good library for using Redis.

HTML5 (and other web-based components) for the UI, because:

- It does not require any installation for the user,
- Consuming REST APIs from the browser using the jQuery^{vi} library is well documented,
- There are good free templates^{vii} to choose from.

There are a few more libraries used in Elastic-BPM, which are explained in detail in the following section where the different components, or Microservices, that make up Elastic-BPM are listed and implemented.

ⁱ NodeJS webpage: nodejs.org/en/

ⁱⁱ The official NodeJS docker image: hub.docker.com/_/node/

ⁱⁱⁱ The express library can be found here: expressjs.com/

^{iv} The library used for consuming REST APIs: github.com/aacerox/node-rest-client

^v The Socket.io webpage: socket.io/

^{vi} The jQuery webpage: jquery.com/

^{vii} Free HTML5 templates: html5up.net/, colorlib.com/wp/free-html5-admin-dashboard-templates/

6.4. Implementing the components

Elastic-BPM is a working minimal viable product (MVP)[5] designed with the following goals in mind:

- to experiment with various BPaaS scheduling techniques,
- to prove that the reference architecture is a viable model with practical advantages, and,
- to demonstrate the scalability and ease of use the Microservices architectural pattern gives when implementing new and cutting-edge solutions.

The following sections provide information on the specific components that together form the BPaaS research system: Elastic-BPM. These components are all open-source and can be found on the Elastic-BPM account on Githubⁱ.

Data storage

Data can be stored in memory or on disk. For backup and disaster recovery, persistent storage at least at some point, is preferred for production systems.

Elastic-BPM makes use of Redisⁱⁱ for data storage. Redis is an open source, in memory data structure store. [69]ⁱⁱⁱ This makes the storage blazing fast and great to work with during experiments. Due to the fact that there is no permanent storage in the Redis container, after each test run the container is removed and a new Redis store is started to make sure there is no residual data from earlier test runs. It is possible to persist the Redis data to disk, in various ways. For more information, I would like to refer the reader to the Redis documentation regarding persistence^{iv}.

While it is not required for the purpose of Elastic-BPM, Redis can also be run in clustered mode making it a high-available in memory data structure store with the ability to scale up to thousands of nodes for free^v. Redis, or Redis-compatible services, are also available at cloud providers, for example Amazon offers Amazon ElastiCache^{vi}, Microsoft offers Azure Redis Cache^{vii}, where Google offers two single-click virtual machine templates: one for a single Redis instance, and one for Redis Cluster^{viii}.

For Elastic-BPM, the official Redis Docker image^{ix} available at the Docker hub, serves our needs. The Command used for running the single Redis container is:

```
docker run --name elastic-redis -d -p 6379:6379 redis
```

Communication

Communication is an essential part in distributed systems. For Elastic-BPM several forms of communication are used, depending on the requirements and the nature of the communicating parties.

ⁱ Github location for Elastic-BPM: github.com/elastic-bpm

ⁱⁱ Redit webpage: redis.io/

ⁱⁱⁱ Redis In Action, free to read online: redislabs.com/resources/ebook/

^{iv} Redis documentation on persistence: redis.io/topics/persistence

^v Scaling Redis: redis.io/topics/cluster-spec

^{vi} Amazon ElastiCache: aws.amazon.com/elasticache/

^{vii} Microsoft Redis Cache: azure.microsoft.com/en-us/services/cache/

^{viii} Google Redis Cluster: console.cloud.google.com/launcher?q=redis

^{ix} Official Redis Docker image: hub.docker.com/_/redis/

For retrieving data from other components in the Elastic-BPM system, REST APIs are used. REST, or Representational State Transfer, is introduced by Roy Fielding in his dissertation. [70] This is widely considered the successor to the industry standard SOAP.

Gathering metrics

As described in [71], when creating a distributed system, it is vital to monitor how the components are performing. In contrast with the binary status of a single application, i.e. it either works or not, multiple moving components can each fail independently and cause a domino-effect bringing the whole system down.

Step eleven of the twelve-factors described in section 6.2: “treat logs as event streams” is great for logging, especially when the number of components approaches or exceeds the number of fingers on one hand.

We use Elasticsearchⁱ for storing logs. The Elastic-stack also offers a monitoring interface called Kibanaⁱⁱ, using which all logs can be inspected and filters can be applied to single out for example errors and failure. Combining Elasticsearch and Kibana with the transformation provided by Logstashⁱⁱⁱ, offers a full solution for monitoring the Elastic-BPM components.

All Elastic-BPM components are run using Docker, the logs are sent to Elasticsearch using the Gelf-driver for Docker^{iv}, in combination with a correctly configured Logstash listener.

User interface and interaction

The user interface is, as stated in the introduction to section 6.4, created using html5. This is a flexible choice, as it is multi-platform and the user can access the system using a browser. The user interface component of the Elastic-BPM system is called elastic-dashboard.

The dashboard is built on top of the “Startmin”^v template, created by Christian Neff and released under the Apache 2.0 license. Some extra logic has been added to split the source code of the page into multiple files and dynamically load the relevant parts of the site. Most notably, Angular^{vi} has been used, which is released under the MIT license^{vii}.

The DataTables^{viii} plugin, part of the Startmin template, has been used extensively to give a live update on the state of the worker nodes and the workflows in the Elastic-BPM system. The most notable part of this Javascript library to display tables, is the possibility to load and re-load data from outside the website using asynchronous web calls. Figure 6.4 shows the usage of a DataTable in the Bootstrap visual template. The DataTable library is available under the MIT license.

ⁱ The ElasticSearch webpage: elastic.co/

ⁱⁱ Documentation on Kibana: elastic.co/products/kibana

ⁱⁱⁱ Documentation on Logstash: elastic.co/products/logstash

^{iv} Documentation on the Gelf driver: docs.docker.com/engine/admin/logging/overview/

^v The Startmin template on Github: github.com/secondtruth/startmin

^{vi} The Angular webpage: angular.io/

^{vii} Wikipedia page on the MIT license: en.wikipedia.org/wiki/MIT_License

^{viii} Webpage for the DataTables plugin: datatables.net/

Running Workflows									
created	id	name	owner	info	status	todo	busy	done	actions
12-11-2016 21:21:11	8d50f800	TestFlow2	johannes	Graph	Busy	0	1	3	
12-11-2016 21:21:01	875a2ca0	TestFlow1	johannes	Graph	Done	0	0	4	
12-11-2016 21:09:18	e442bab0	TestFlow-Human-2	johannes	Graph	Waiting	2	0	2	
12-11-2016 21:09:08	de4c1660	TestFlow-Human-1	johannes	Graph	Waiting	2	0	2	

Figure 6.4: Screenshot of a live-updated DataTable.

For displaying the BPM graph, the Cytoscape.js libraryⁱ is used. This library can also be used for network analysis, but is currently only used for displaying the DAG graph and coloring the nodes according to state. Figure 6.5 shows an example flow, visualized using the Cytoscape.js library. This library is available under the MIT license.

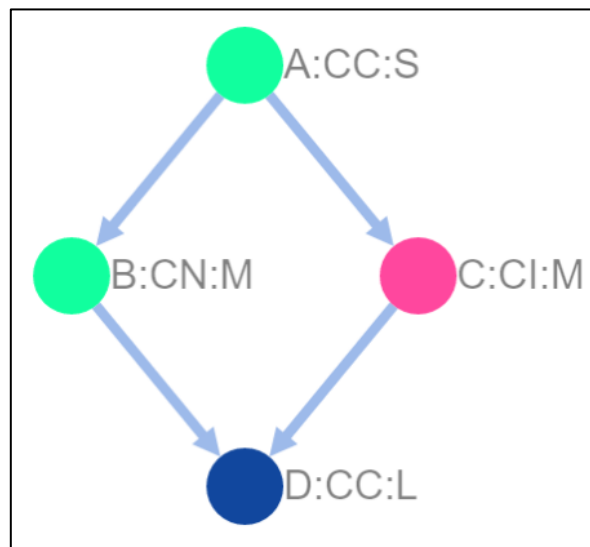


Figure 6.5: Test flow, visualized using Cytoscape.js.

Administration and configuration

There are multiple components that have to be hosted and monitored when building a microservice platform. Component discovery is also required: components need to be able to communicate with each other.

The Docker-Compose toolⁱⁱ, available for free from Docker.com, is able to build and run multiple Docker containers and facilitate component discovery. The docker-compose.yml fileⁱⁱⁱ used to run the Elastic-BPM system is visible at the Elastic-BPM Github page. Using NPM^{iv}, the Node.js package manager, it is possible

ⁱ Webpage on the Cytoscape library: js.cytoscape.org/

ⁱⁱ Documentation on the Docker-Compose tool: docs.docker.com/compose/

ⁱⁱⁱ Documentation on the Docker-Compose file: docs.docker.com/compose/compose-file/

^{iv} Webpage for the Node Package Manager: npmjs.com/

to script the creation of required Docker components in the background, while only having to focus on the component being written.

When running the system in the test environment, logs are sent to an Elasticsearch environment to analyze using Log4jsⁱ and Gelfⁱⁱ.

Workflow component

The workflow component stores all data on the workflows directly into the Redis repository. This makes the component simple and stateless in nature, while for Elastic-BPM only a single workflow component is live at runtime, it is possible to scale out horizontally using this architecture.

The nodes of a workflow are stored using a string with comma separated identifiers. The edges of the workflow graph are stored using a comma separated tuples of nodes, separated by the '->' sign. This notation describes the directed graph.

The identifiers of the nodes determine the type and size of the task. Each identifier consists of three parts, separated by a colon sign. The first part is a unique name, the second part is the type of the task, the third part of the node identifier is the size

The workflow component allows the retrieval of a single workflow using the unique identifier, the retrieval of all workflows, updating a workflow, deleting a workflow, deleting all workflows, and, the addition of a set of workflows using a JSON script file. This file also contains a delay parameter per workflow, defining how much time to wait between the submission of the script and the moment to add the workflow to Elastic-BPM. This feature is used in our experiments, emulating a real-life workload.

Currently there is no way to change the flow of a workflow once it has been entered into the Elastic-BPM system. This is part of future work, as described in section 8.2.

Because of the scaling in a BPaaS environment, it can happen that the execution of a task is pre-empted due to the node that the worker is running on gets deallocated. In this case, the work on the task is restarted by a different worker component, probably on a different node a while later. For this reason, there is a timeout set in the scheduler component. When the timeout fires, Elastic-BPM assumes the worker previously working on this task is stopped due to the machine being deallocated, and the task is flagged as 'todo' again. This gives other worker components the ability to pick it up and finish the task.

In the event of two or more worker components working on the same task, the first one to finish saves the results. The other workers submitting results for tasks that have already been flagged as 'done' get an error when trying to submit the task to the scheduler, they will then start work on a new task.

Scheduling component

The scheduler component handles the logic in the selection of tasks to execute, the logic for scaling the number of nodes in the system, and the reporting of the metrics it is responsible for.

When a task is called for, either by an instance of the worker component, or a human consumer, the scheduler collects the tasks available for the consumer and selects a task to return. This functionality is exposed via the '/tasks/worker/free' and '/tasks/human/free' API calls, which the worker and human component can call when they are finished with their previous task – after declaring the previous work done using the update API calls.

ⁱ Github location of the Log4js library: github.com/stritti/log4js

ⁱⁱ Documentation on the Gelf driver: docs.graylog.org/en/2.2/pages/gelf.html

In section 5.2, the Static, On-Demand and Learning policies have been introduced. These are implemented in the *resource manager* part of the scheduler component. The policies all follow the same principle: first gather information on the running system, then determine the changes required to get to the desired state, and finally execute these changes.

The gathering of information on the running system and the execution of the changes on this system, are made possible by the scaling component, which is described in the following section.

Scaling component

For each implementation of Elastic-BPM, there is a set of 16 virtual machines created in the Microsoft Azure cloud using a DevTest Labⁱ. The machines used for Elastic-BPM are created using the “Basic A1” size, which at the time of writing has the following technical specifications:

Number of virtual cores	1
Memory	1.75 GB
Disk size	40 GB

This is a relatively small virtual machine, capable of running the Docker engine which is used to execute the Worker component. The advantage of using small machines, is that CPU, memory and IO load is quickly visible. More information on this setup can be found in section 7.2.

Starting and stopping virtual machines is achieved using the `azure-cli` from within Node.js. These commands are executed using the *cross-spawn* Node.js packageⁱⁱ, which executes commands in the command-line and makes it possible to capture the resulting output in Node.js.

Each machine can be stopped by issuing the command:

```
azure vm deallocate <resourcegroup> <virtualmachine>
```

where the `<resourcegroup>` and `<virtualmachine>` variables define which machine needs to be stopped (deallocated). Starting a machine can be achieved by issuing the command:

```
azure vm start <resourcegroup> <virtualmachine>
```

The Elastic-BPM Scaling component makes it possible to login to Microsoft Azure, and start and stop machines using a simple API interface the Scheduler component or system operator can call. Extending this component to use other cloud providers or to offer different calls, is straight-forward due to the low complexity involved in executing command-line statements.

Docker component

The Docker component is created to start, stop and scale the Docker serviceⁱⁱⁱ, running the Worker components. All nodes described in the previous section, are connected using Docker swarm mode^{iv}. Docker swarm mode enables these machines to act as a single entity and enables Elastic-BPM to start the worker components on each node with ease. To see which nodes are connected and what the status is of

ⁱ Documentation on the DevTest Labs: azure.microsoft.com/en-us/services/devtest-lab/

ⁱⁱ The cross-spawn Node library: npmjs.com/package/cross-spawn

ⁱⁱⁱ Documentation on Docker Swarm services: docs.docker.com/engine/swarm/services/

^{iv} Documentation on Docker Swarm mode: docs.docker.com/engine/swarm/

the nodes, it is possible to run the 'node ls'ⁱ command on the master node. Figure 6.6 shows the output of this command with 11 machines in a swarm.

```
johannes@master-01:~$ sudo docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
38rqcj22i8m6fm9jff0h6cva2y	node-01	Ready	Active	
5r8b10klye0s7osmuan6720mp	node-06	Down	Active	
5vpk2osjvcw1n39giiq6332gj	node-02	Down	Active	
7dc4hswfj4cnitrresnyh66zx	node-09	Down	Active	
a96pq9o9ybsyn4skyj7vmfxol	node-07	Down	Active	
b0y43g5rxfvv9pggt8t7rdhrn	node-08	Down	Active	
beijdnz2is152fcg21ljzy2wj	node-10	Down	Active	
dywp8yib10fc6j9hzhfz612f5	node-04	Down	Active	
e3yhxb2goi8tuh0uym8oe8ko	node-03	Down	Active	
ee3yp0xst04aqymdpxwf8s7vc	node-05	Down	Active	
eg1m38x0etrmxwce1b55yj6kx *	master-01	Ready	Active	Leader

```
johannes@master-01:~$
```

Figure 6.6: Output when running the Docker 'node ls' command.

Worker component

The worker component consists of a relatively simple script that repeatedly executes the following steps:

1. Get a task executable by this worker from the scheduler component
2. Execute the task
3. Signal the scheduler component it is ready

If there is no task currently available for the worker component or the scheduler component is not available, it sits idle for a pre-defined amount of time and then queries the scheduler component again. All of these actions are also logged to the console, which in turn is sent to the Elasticsearch engine for analysis.

As this is a research BPaaS instance, Elastic-BPM only has a limited set of tasks to execute. These tasks are listed in section 0, which describes the workflow component.

For the CPU intensive task, task CC, the first N primes are calculated. The size of N depends on the size of the task: for tasks denoted with S (small), N equals 400.000, for tasks denoted with M (medium), N equals 800.000, for tasks denoted with L (large), N equals 1.600.000.

The following Javascript function is used to determine if a given n is prime or not:

```
isPrime = function(n) {
    if (n < 2 || n !== Math.round(n)) return false;
    q = Math.sqrt(n);
    for (var i = 2; i <= q; i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return true;
};
```

ⁱ Documentation on the 'node ls' command: docs.docker.com/engine/reference/commandline/node_ls/

For the network intensive task, task CN, a file of 100MB is downloaded. This file is stored in an Azure storage account, accessible by HTTP. The *node-rest-client*ⁱ is used to download this file. The amount of times the file is downloaded depends on the size of the task. Once for S, twice for M, five times for L.

For the IO-intensive task, task CI, large files containing random strings are written to disk. For the generation of the random strings, the *randomstring*ⁱⁱ package is used. The amount of random strings written to disk depends on the size of the task. 100.000 x 32 characters for S, 400.000 x 32 characters for M, 200.000 x 32 characters for L.

The worker component is the component executing all the system executable tasks in a workflow, the human executable tasks are run either by humans in a production scenario, or by the simulated human component for experiments. This component is described in Chapter 7, where we validate the BPaaS implementation.

6.5. Summary

In this Chapter we have described the implementation of Elastic-BPM, our prototype implementation of BPaaS based on the reference architecture presented in Chapter 4. For this implementation, the use of Docker, part of the Agile way of working and microservices patterns have been used.

The implementation of Elastic-BPM is the technical contribution of this Chapter.

ⁱ The REST client used for Node: npmjs.com/package/node-rest-client

ⁱⁱ The library used for generating random strings: npmjs.com/package/randomstring

7. Experimental Validation of the Elastic-BPM Prototype

Elastic-BPM is created to demonstrate the possibilities with the BPaaS reference architecture. In this Chapter, we show the characteristics of this system by executing test-flows multiple times and describing the results.

We also put the provisioning policies to the test and find, through rigorous experimentation, which combination of policy and settings performs best under a large set of circumstances.

The results of our experiments, and specifically the learnings, are our scientific contributions in the field of provisioning a BPaaS system with humans in the loop.

7.1. Experiments, Workloads and Metrics

Table 7.1 shows the research goals and corresponding experiments we will run. Each experiment focusses on a different aspect of the Elastic-BPM system, ranging from the impact of the system resources, the scaling parameters, to the human workforce characteristics.

Goal ⁱ	policy	#nodes	#threads/ node	threshold	#humans	schedule ⁱⁱ	workload	reps	total runs
a	Static	10	[1 upto 8]	-	6	i	wl1	1	8
b	Static	1 to 15 ⁱⁱⁱ	Best of a	-	6	i	wl1	1	8
c	Static	10	Best of a	-	6	i	[wl1, wl2]	1	2
d	OnDemand	min 1, max 15	Best of a	0.2 to 0.55 ^{iv}	6	i	wl1	1	8
e	Learning	min 1, max 15	Best of a	15 to 50 ^v	6	i	wl1	1	8
f	Learning	min 2, max 15	Best of a	Best of e	3 to 24 ^{vi}	i	wl1	1	8
g	Learning	min 2, max 15	Best of a	Best of e	9	[i, ii]	wl1	1	2
h	Learning	min 2, max 15	Best of a	Best of e	9	i	wl1	10	10

Table 7.1: Experiments to run

ⁱGoals:

- Impact of the amount of threads per machine on the performance of the static system *using Static*
- Impact of *the number of machines on the performance of the static system using Static*
- Impact of *the workload on the performance of the static system using Static*
- Impact of *the scaling threshold on the performance of the system using OnDemand*
- Impact of *the scaling threshold on the performance of the system using Learning*
- Impact of *the number of humans on the performance of the system using Learning*
- Impact of the schedule of the humans on the performance of the system *using Learning*
- Stability of the system over 10 runs

ⁱⁱ Human schedules:

- 24-hour workday for x workers
 - 0h-8h: x/3 workers
 - 8h-16h: x/3 workers
 - 16h-24h: x/3 workers
- 8-hour workday for x workers
 - 0h-8h: 0 workers
 - 8h-16h: x workers
 - 16h-24h: 0 workers

ⁱⁱⁱ Increased with steps of 2

^{iv} Displayed threshold is the lower bound, increased with steps of 0.05. The upper bound is 0.2 higher.

^v Displayed threshold is the lower bound, increased with steps of 5. The upper bound is 10 higher.

^{vi} Increased with steps of 3

Workloads are generated using a specific workload generatorⁱ, which generates workloads according to a set of input parameters. These parameters include *number of workflows*, *duration*, and, *random seed*. The random seed, in combination with the *random-js*ⁱⁱ package, enables the workflow-generator to generate a random workflow-list. This list can be re-created using the same random seed and input parameters.

The duration argument determines the highest delay value of the workflow can have. A duration argument of 30 translates to a maximum delay of 30 minutes, or 1.800.000 milliseconds, for any of the generated workflows. The amount argument determines how many workflows are generated.

The workflow-generator has 3 types of scripts that are added randomly to the workflow-list. The type 1 and type 2 workflows consist of system tasks only, while the type 3 workflow also contains 'human in the loop' tasks in the workflow. A random selection of these types of workflows are added to workload, and are each given a delay between 0 and the specified duration argument. This workload is then written to the specified JSON output file.

We define a workload consisting of 50 workflows as "Regular load" (wl1), and a workload consisting of 100 workflows added to the system in the same amount of time is defined as "High Load" (wl2). Figure 7.1 shows us a graphical layout of the workloads and the type of workflows added over timeⁱⁱⁱ. In this figure, the job arrival times for the 3 different types of workflows is shown for both the regular load and high load workload. For example, the job arrivals for the Regular workload (wl1), which is depicted at the top of the figure, start with one arrival at time 0, followed by a cluster of arrivals around time=5 minutes, with a few more sparse arrivals until the end of the workload.

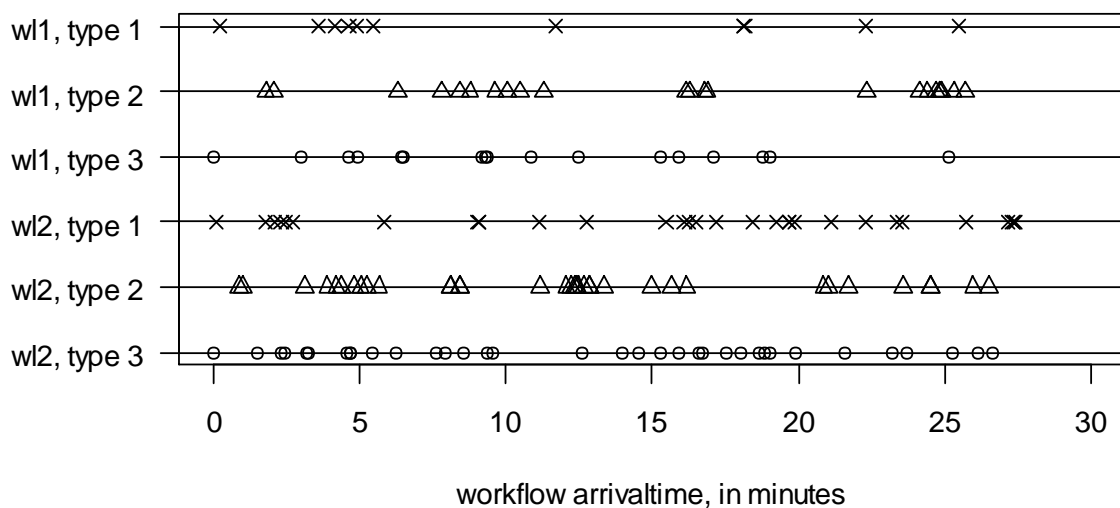


Figure 7.1: Arrival times for wl1 (top 3) and wl2 (bottom 3), subdivided per type of workflow.

ⁱ Workload generator on Github: github.com/elastic-bpm/workflow-generator

ⁱⁱ Documentation on the random package: npmjs.com/package/random-js

ⁱⁱⁱ Note that while the theoretical maximum of 30 minutes is defined, there is (in this case) no workflow that has this delay, the realized maximum delay is around the 28 minutes.

7.2. Running the experiment

A Docker Swarm environment with a single head VM and 15 worker VMs is used to execute the worker components. All of these VMs are provisioned in the same DevTest labⁱ on Azure, running in the ‘West Europe’ datacenter.

Each VM is sized “Basic A1” (1 cores, 1.75 GB memory)ⁱⁱ. Microsoft Azure makes a distinction between the Basic and Standard tier. The Basic tier is only available up to A4, has slightly lower Disk IO than the Standard tier and does not allow (built-in) load-balancing and auto-scalingⁱⁱⁱ. The cost, \$23 per month, and simplicity of the “Basic A1” machines, makes it a perfect choice for our experimental setup.

For the calculation of the costs, we will be using a cost of \$0.0005 per minute per node, which is roughly equal to \$0.03 per hour and \$23 per month if run continuously.

Specification for this Basic A1 machine:

CPU cores	Memory: GiB	Max data disk throughput: IOPS	Max NICs	Network bandwidth
1	1.75	300	1	moderate

The entire test is run on 4 separate Azure subscriptions, with 4 identically provisioned DevTest labs. All components and machines running in these labs are configured to log to different instances of Elasticsearch, which is explained in detail in section 6.4.

At the start of each test, the machines in the DevTest lab are in a powered-off state (Deallocated^{iv}). The following steps are executed in order for a single run of an experiment:

1. All machines are started using the Azure SDK
2. If there is any Docker component still active, it is shut down
3. Any present source code is removed
4. The actual source code is cloned from the public Github repository
5. From this source code, the components are built using docker-compose
6. The system is started using docker-compose
7. Once the system is available, there is a 5-minute wait for machines to settle down
8. The ‘runTest’ API endpoint is called with a payload specific for the current test run
9. The ‘testDone’ API endpoint is polled, until this gives a positive response
10. Once the test is done, all worker components are removed
11. The policy is set to ‘Off’
12. The system is brought down and removed using docker-compose down
13. All machines are stopped (Deallocated)
14. From Elasticsearch, all application logs and metrics are downloaded

These steps are automated and executed using shell-scripts (available as open source on Github^v), the Node package manager Yarn^{vi}, and SSH^{vii} for executing scripts remotely.

ⁱ Microsoft Azure DevTest lab: azure.microsoft.com/en-us/services/devtest-lab/

ⁱⁱ Documentation on Azure VM A-sizes: docs.microsoft.com/en-us/azure/virtual-machines/windows-sizes#a-series

ⁱⁱⁱ Azure Basic VM-sizes: azure.microsoft.com/en-us/blog/basic-tier-virtual-machines-2/

^{iv} Documentation on the Deallocated state: blogs.technet.microsoft.com/gbanin/2015/04/22/difference-between-the-states-of-azure-virtual-machines-stopped-and-stopped-deallocated/

^v Location of the execution scripts for these experiments: github.com/elastic-bpm/elastic-kickstart

^{vi} Yarn homepage: yarnpkg.com/en/

^{vii} Wikipedia on SSH: en.wikipedia.org/wiki/Secure_Shell

Human Component

For running the experiments, we introduce the human component. This component is only required for running simulations; in a production environment, the actual humans interacting with the BPaaS system are required to perform the human-oriented tasks in workflows.

In section 5.2, when describing the workflow component, 5 different type of tasks are described. The tasks with the **HE** and **HH** identifiers are to be completed by humans, these are the tasks this human component executes. In contrast to the worker component, described in section 6.4, the “work” aspect of the human component is implemented by the *setTimeout*ⁱ function in Javascript. This function is one of the native timer functions in Javascript and takes two arguments: the first is the action to perform when the timer runs out and the second is the number of milliseconds to wait.

The human component can simulate a large number of humans working at the same time, due to the async nature of the *setTimeout* command used to simulate the work. A work schedule is implemented with four variables: *on-*, *off-*, *init-*, and *total-time*. The result of such a schedule can be seen in Figure 7.2, here the *on-time* is 8 hours, with 16 hours *off-time*. The first cycle starts after 8 hours (*init-time*) and the full schedule takes 72 hours (*total-time*).

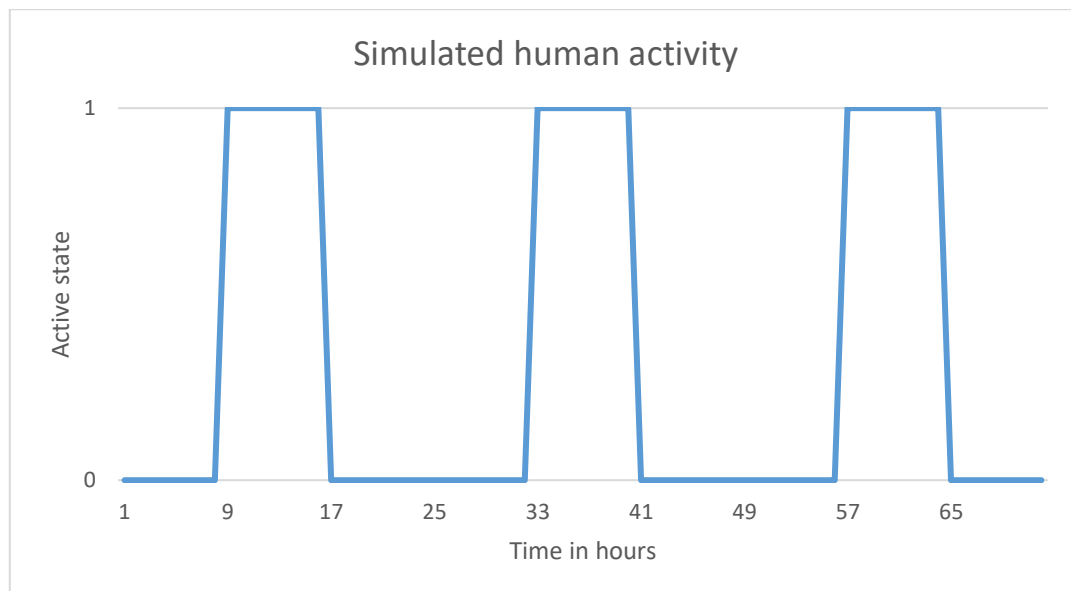


Figure 7.2: Work schedule ii for simulated humans.

There are three main differences between the worker and human components. The first difference is the type of tasks they pick up from the workflows: the worker component takes tasks starting with the C (for compute) identifier, the human component takes tasks starting with the H (for human) identifier. The second difference is the way the tasks are executed: the worker component performs calculations, network- or disk-operations, the human component sets a timeout. The third difference is the fact that humans are simulated using a work schedule defining the human activity, where the worker component is always active.

ⁱ Node timer functions: nodejs.org/api/timers.html

7.3. Results and recommendations

During the performed experiments, all important data is transmitted to the Elasticsearch clusters using two indices: logging data from the application log and metric data from the running nodes themselves, comparable to the approach by [72]. This data is then analyzed using the R statistical programming language. [73]

The runs each take around 2400 seconds (40 minutes), after which the humans are finished with their schedule and all nodes are shut down. One minute in the test run simulates a full hour, for example: the humans work 8 of every 24 minutes during the test, which corresponds with a full work shift (8 hours) every day (24 hours). The full set of results, including the raw logs and metric data, have been made available for downloadⁱ.

User-oriented evaluation metrics

Table 7.2 shows the user-oriented metrics we measure for each completed business process.

Metric	Description
Makespan T_m	the time between the first start of the business process and the finish
Execution time T_e	the sum of all runtimes of the tasks within the business process
Wait time T_w	the time before the business process gets picked up by the system
Response time T_r	the sum of the wait time and the makespan: $T_r = T_w + T_m$
Human delay time T_{dh}	the amount of time a business process is delayed by human tasks
Human time T_h	the time spent by humans in the process
System delay time T_{ds}	the amount of time a business process is delayed by system tasks
System time T_s	the difference between the <i>response time</i> and <i>human time</i> : $T_s = T_r - T_h$

Table 7.2: User-oriented metrics measured for each completed business process.

Makespan, *execution time*, *wait time* and *response time* are introduced in [52], *human delay time*, *human time*, *system delay time* and *system time* are original and specific to this research.

Because the *wait time* metric does not show us which part of the slowdown is due to the humans in the loop, we introduce *human time*. When we subtract the *human time* from the *response time*, this gives us a fair metric for speed of the system: *system time*.

System time, T_s , is the main focus of our research. We optimize the system around the user input.

Cost metrics

Costs are an interesting topic, always. When not using cloud servers and pay-per-use pricing, there is a difference between initial costs and running costs. The former is commonly known as capital expenditure or CAPEX. The latter as operational expenditure or OPEX.ⁱⁱ As our system is fully *cloud native* [74], there is no CAPEX (unless you are using a cloud service provider which uses a different cost-model) and all costs are OPEX or running costs. We therefore define only the following metric for cost:

Cost: Spend for a full run of a simulation, specified in node-minutes and dollars charged to our credit-card.ⁱⁱⁱ

This *cost* does not include the base machine(s) used for hosting the system itself and the log-collection server as this is equal for each run of the experiment.

ⁱ Link to results: <http://bit.do/bpaas-data>

ⁱⁱ CAPEX and OPEX explained in more detail: diffen.com/difference/Capex_vs_Opex

ⁱⁱⁱ In our case this is a virtual credit-card as the simulation is run on (free) Azure credits

Results for Experiment A

The goal of experiment a, as shown in section 7.1, is to determine the impact of the amount of threads per machine on the performance of the static system. The system is put under test in 8 distinct runs, each run using a different number of threads per node, starting at 1 thread per node for the first run up to 8 threads per node for the final run. The human component is set according to the working hours schedule, as it is for each experiment except for experiment G. Results from experiment A are shown in Figure 7.3 and Figure 7.4.

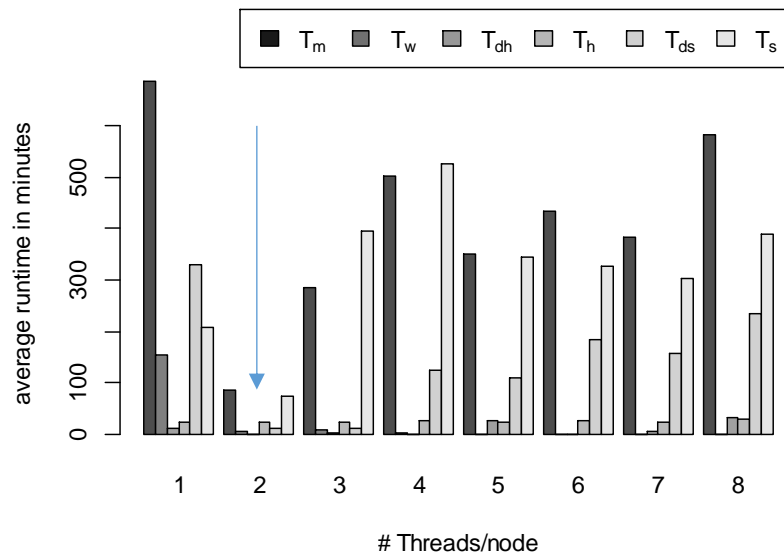


Figure 7.3: Absolute runtime of the static algorithm, with a different number of threads per run

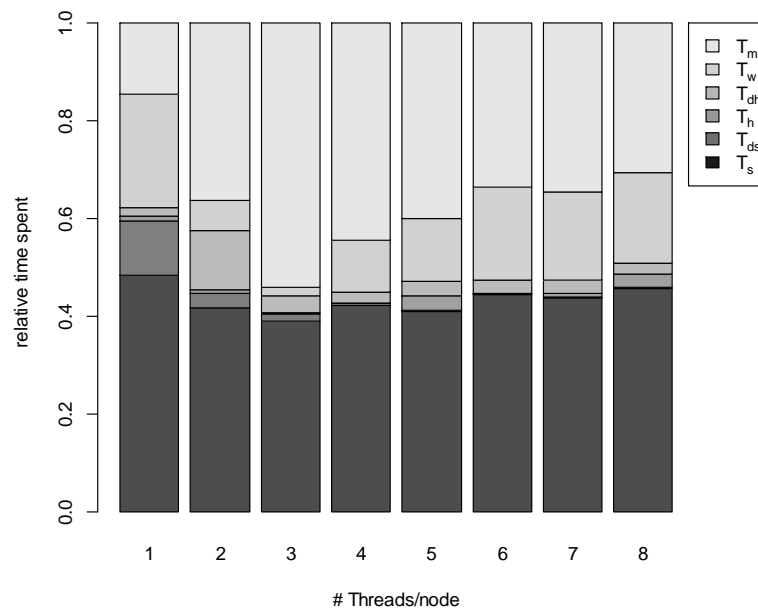


Figure 7.4: Relative runtime of the static algorithm, with a different number of threads per run

Figure 7.3 shows us that the absolute runtimes (all of them, in this case) are much lower when using 2 threads as compared to the other runs, this is indicated by the blue arrow. When running more than 3 threads per node, the system becomes unstable due to the amount of memory available and does not

finish the workload during the time allotted. We will be using 2 threads per node for all following experiments.

As there are always 10 machines running, the costs for this experiment has been roughly \$12 per run.

Results for Experiment B

The goal of experiment b, as shown in section 7.1, is to determine the impact of the number of machines on the performance of the static system. We execute the workload 8 times, each with a different number of machines available for the static provisioning policy. Each machine runs 2 threads, as was found optimal in the previous experiment.

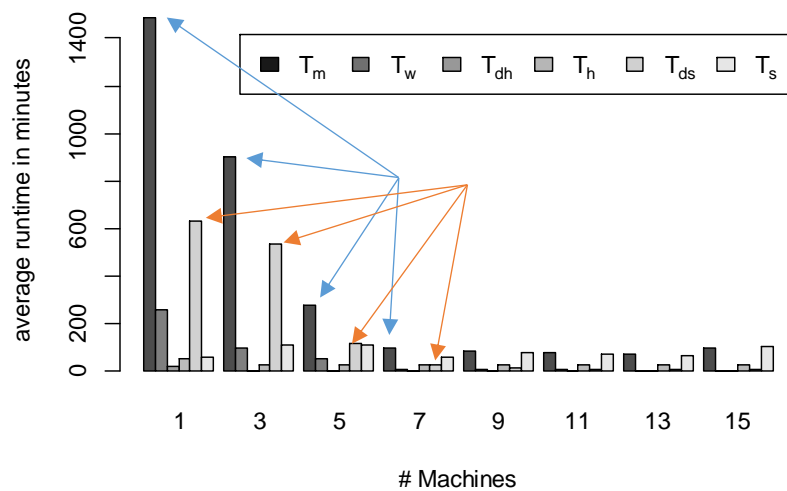


Figure 7.5: Absolute runtime of the static algorithm, with a different number of machines per run.

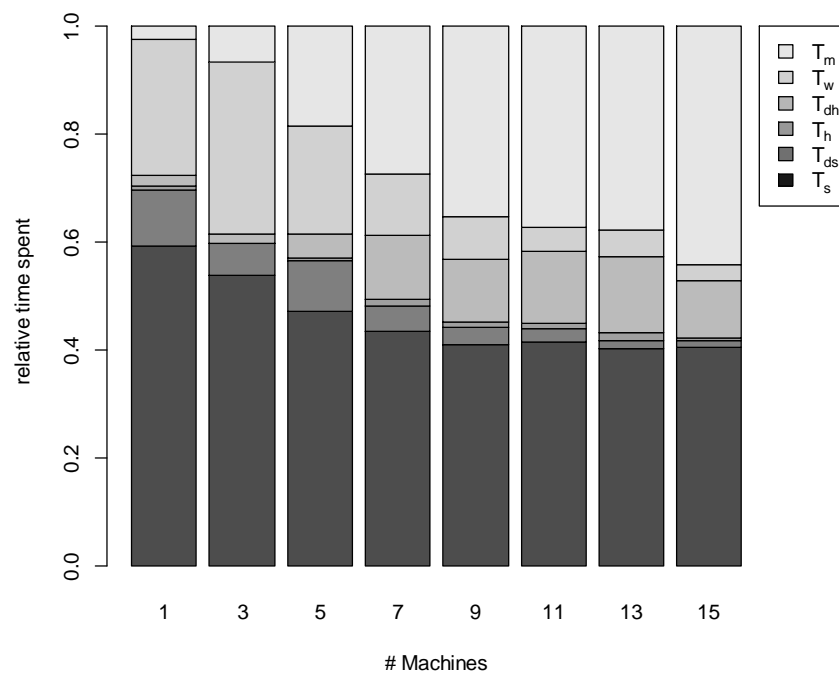


Figure 7.6: Relative runtime of the static algorithm, with a different number of machines per run

Figure 7.5 shows the absolute runtimes, which sharply decrease when the amount of machines increases. This is indicated by blue the arrows in the figure. The delay caused by the system, T_{ds} , decreases sharply as well, when increasing the number of machines. This is indicated by the orange arrows in the figure. These sharp declines diminish when adding more than 7 machines to the system, while using the static provisioning policy.

Using less than 5 machines with the static provisioning algorithm does not finish the workload provided, which makes the presented results even stronger, as only completed workflows are counted in these averages.

These results give us, together with results from the previous experiment, a recommendation for the static provisioning policy to use 2 threads per machine, with 7 machines.

The cost for this experiment has risen linearly from ~\$1.20 for 1 machine to ~\$18 for 15 machines.

Results for Experiment C

The goal of experiment c, as shown in section 7.1, is to see the difference the intensity of the workload itself makes. There are two workloads being tested: one with 50 workflows, one with 100 workflows. Figure 7.1 in section 7.1 shows the contents of these workflows.

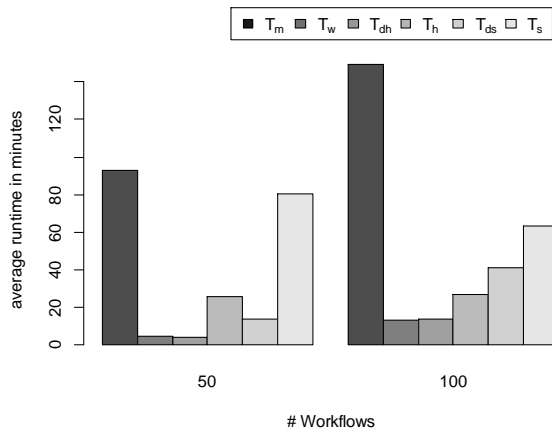


Figure 7.7: Average runtimes for experiment c

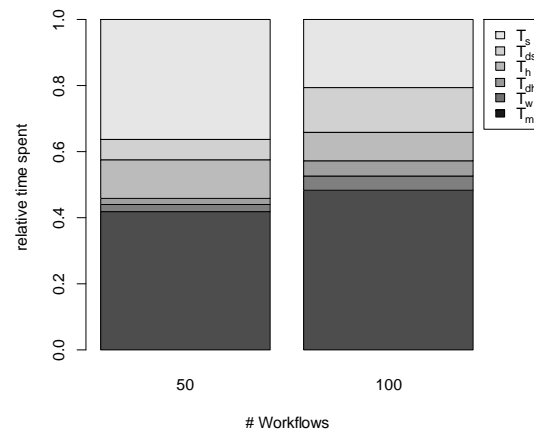


Figure 7.8: Relative runtime for experiment c

Figure 7.7 shows that the average time for the makespan per workflow increases when running 100 workflows (right) instead of 50 (left), when comparing the relative times in Figure 7.8, it is clear that the largest difference is in the delay caused by the system, T_{ds} . This increased delay is caused by the higher average load when executing 100 workflows instead of 50. Figure 7.9 shows the average load when running workload $w/2$ is significantly higher than when running $w/1$. Figure 7.8 shows that the delay caused by the humans, T_{dh} , also increases with a higher workload, but the impact is relatively small.

As both runs have used 10 machines constantly, the costs for each run has been just about \$12.

Results for Experiment D

The goal of experiment d, as shown in section 7.1, is to find the influence of the scaling threshold on the performance of the system using the OnDemand provisioning policy. We examine this influence by increasing the scaling threshold each separate run. For the OnDemand provisioning policy all nodes are available, but will only be used when the scheduler activates them. A new node is activated when one of the active machines crosses the upper load threshold. Any node not recently started which has less load than the set lower threshold is deactivated. The upper load threshold is 0.2 higher than the lower load threshold, for all experiments.

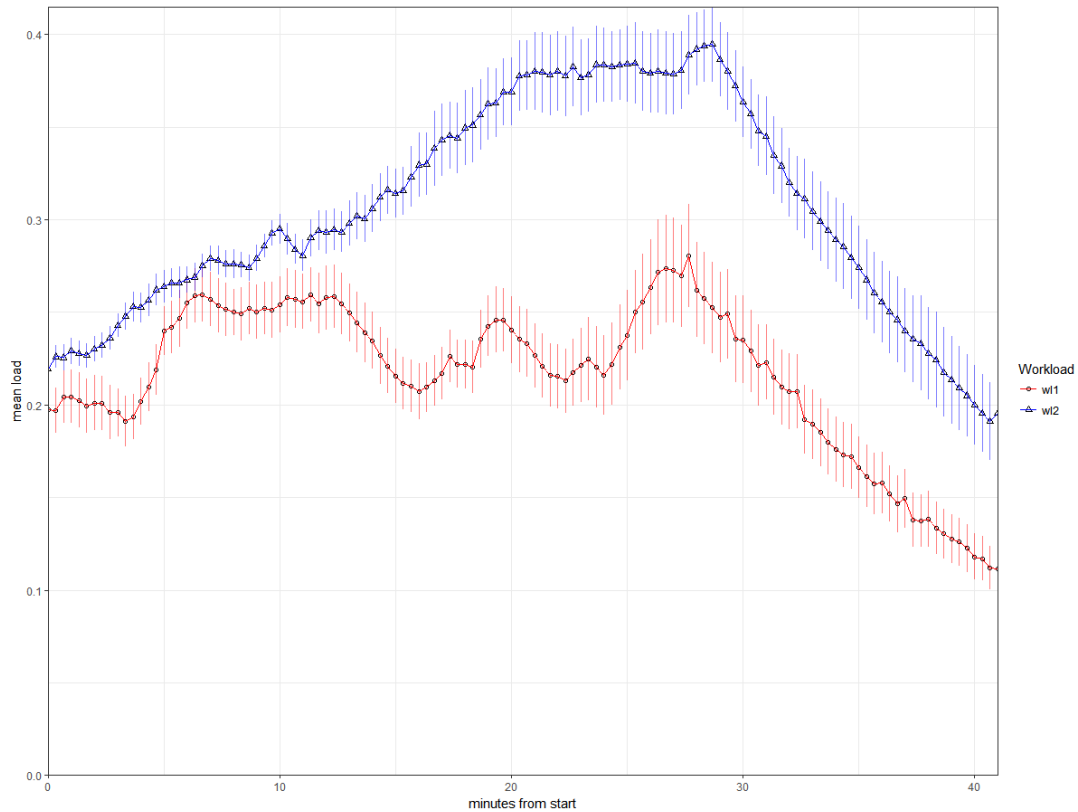


Figure 7.9: the average load of all active nodes for wl1 and wl2 during experiment c

Figure 7.10 shows the average runtimes for different values of the lower threshold while using the OnDemand provisioning policy. Lower values for this scaling parameter give better results. This is due to the fact that a lower value for the threshold results in earlier activation of new nodes. Experiment b has shown that more machines results in lower average runtimes, up to 7 nodes.

When the lower threshold is 0.4 or higher, there are hardly any new nodes activated, causing the system to perform roughly equal to the static provisioning policy with 1 node. As we have concluded in experiment b, running the workload with only 1 node does not give good results, at least 5 are needed to finish the workload.

Table 7.3 shows us the running costs for each run of experiment d. Run 3 was clearly the cheapest successful run from this experiment.

Table 7.3: Running costs for experiment d

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8
\$6.49	\$6.36	\$4.85	\$5.18	\$1.56	\$1.38	\$1.21	\$1.22

Results for Experiment E

The goal of experiment e, as shown in section 7.1, is to find the influence of the scaling threshold on the performance of the system using the Learning provisioning policy. We examine this influence by increasing the scaling threshold each separate run. For the Learning provisioning policy all nodes are available, but will only be used when the scheduler activates them. A new node is activated when there are more remaining tasks queued than is set in the upper threshold. When the number of tasks in this queue decreases below the lower threshold, the least busy node, measured in load, is deactivated. The upper threshold is always 10 higher than the lower threshold.

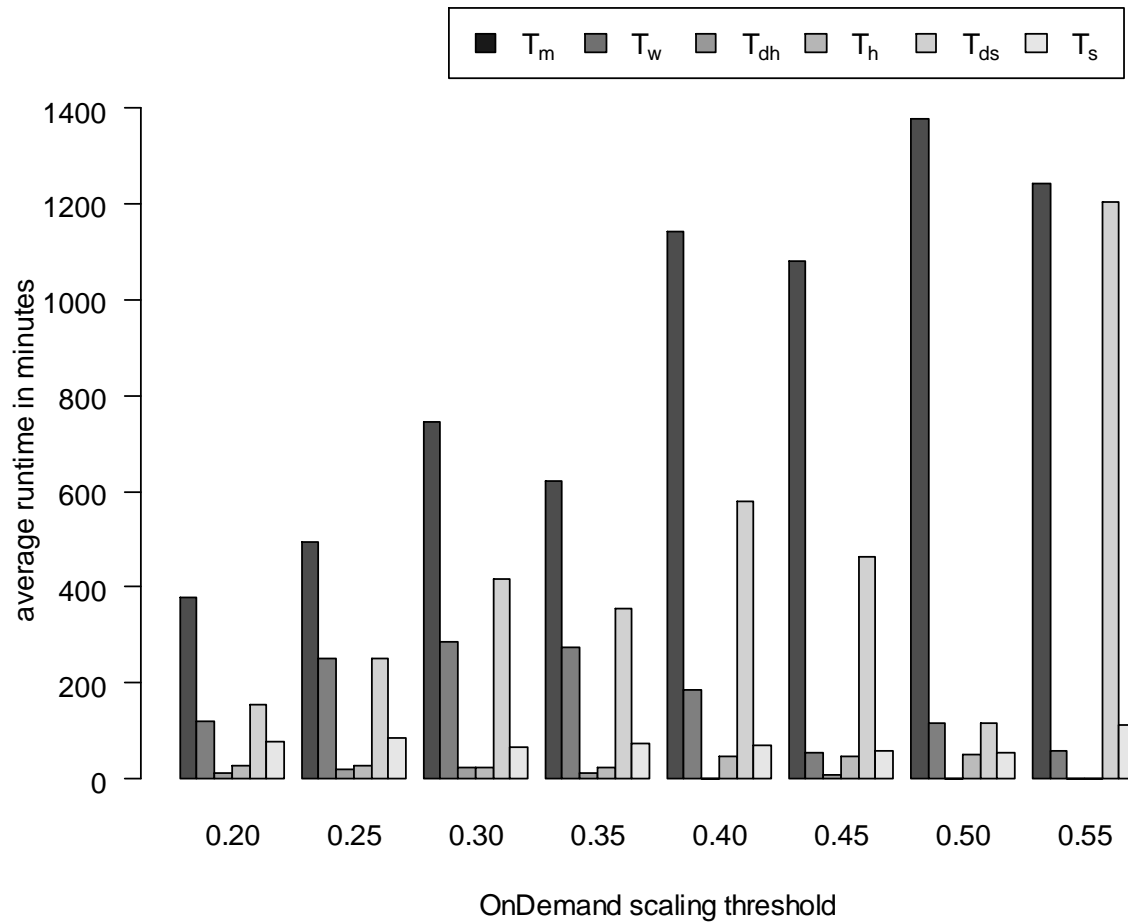


Figure 7.10: Average runtime showing the influence of the OnDemand scaling threshold

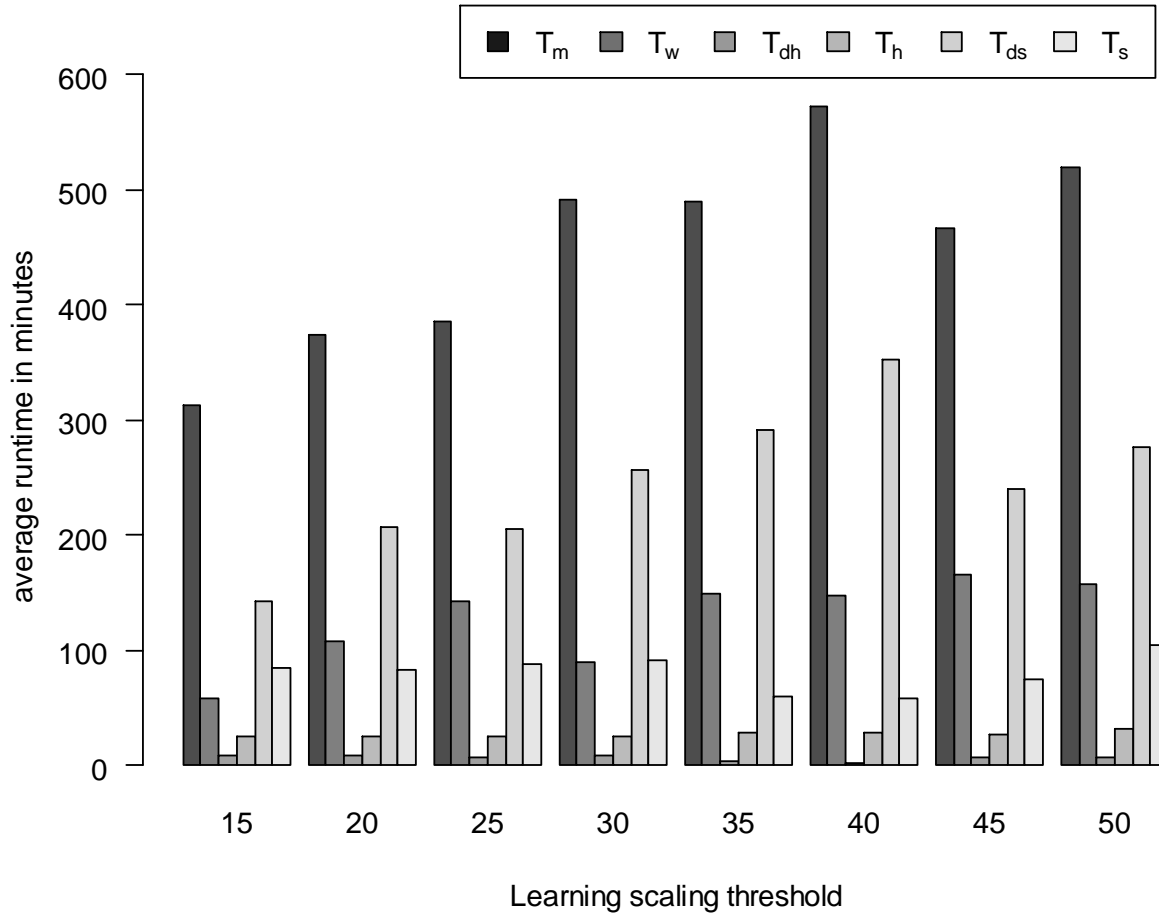


Figure 7.11: Average runtimes for different scaling thresholds for Learning provisioning policy

Figure 7.11 shows the average runtimes for different scaling thresholds for the Learning provisioning policy. Only during runs 1, 2, 3, 4, and, 7 the workload was completed in the allotted time. From these different runs, clearly the first run shows the lowest average runtimes.

The advantage of scaling up machines early with a low scaling threshold, pays off in reduced T_w and T_{ds} , which in turn reduces the overall makespan, T_m . The human-influenced factors T_{dh} and T_h are unaffected, just like the time spent executing system tasks, T_s . As scaling down is also done quickly with a low scaling threshold, and the workload is finished earlier, the costs with a lower scaling threshold are comparable with the runs with a higher threshold.

The costs for the successful runs ranged from \$4.43 for the 3rd run to \$4.74 for the 7th run. The 1st run, with the best average runtimes, had a running cost of \$4.57.

Results for Experiment F

The goal of experiment f, as shown in section 7.1, is to find the influence of the number of humans on the performance of the system using the Learning provisioning policy. We examine this influence by increasing the number of humans executing human tasks each separate run with steps of 3. The first run there will be 3 humans executing tasks, run up to 24 humans the last run.

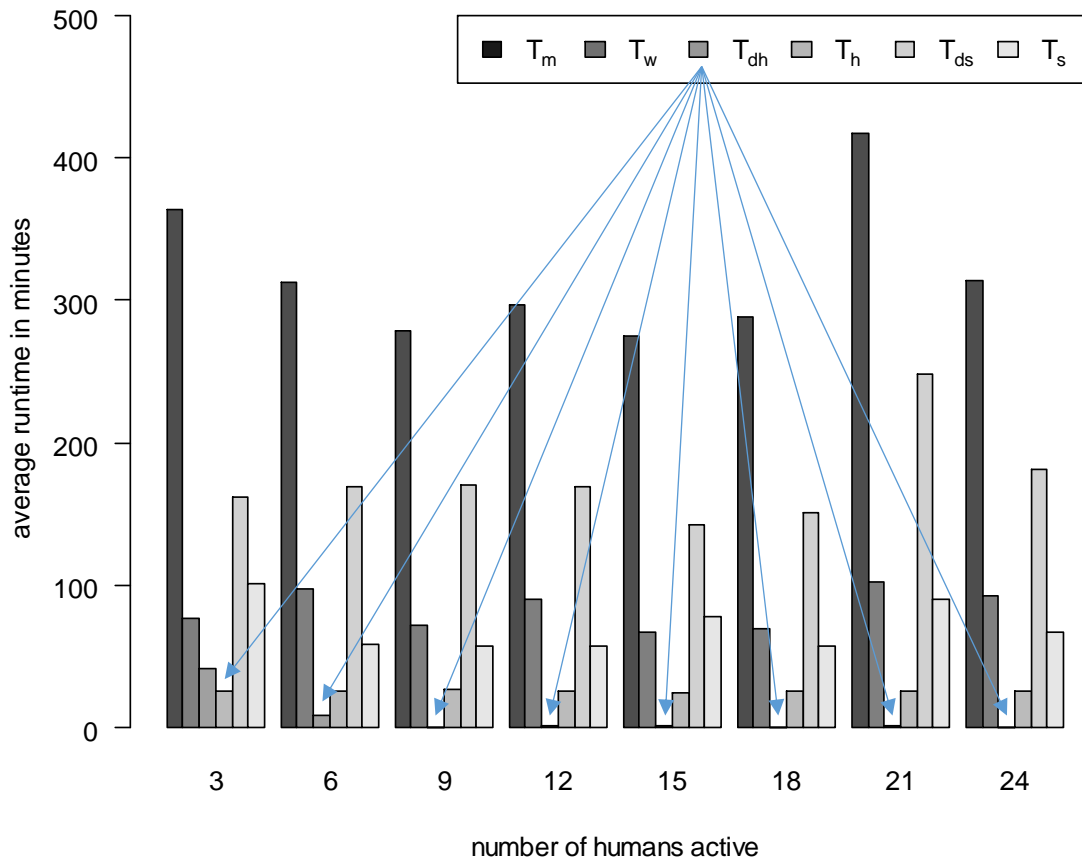


Figure 7.12: Average runtimes with different number of humans in the loop (experiment f)

Figure 7.12 shows the average runtimes for each run of experiment f. From 9 humans up, the human delay factor, T_{dh} , is negligible. The arrows in this figure point to these times, or the lack of them. This result is due to the amount of work available for human workers. When 9 humans or more are actively working on the ‘human in the loop’ components of the workflow, there is an overcapacity, which drastically reduces the T_{dh} as there is a human ready to start with work at any given moment.

The first run of this experiment, with 3 humans in the loop, had a running cost of \$5.65. All other runs had running costs lower than \$4.70. We do not take into account the costs of the humans in this calculation.

Results for Experiment G

The goal of experiment g, as shown in section 7.1, is to determine the influence of the human work schedule on the performance of the system using the Learning provisioning policy. We examine this influence by executing two test runs, each with a different worker schedule. The first schedule is the same schedule we have been using for all previous experiments: the workers are active around the clock. The second schedule the workers are active 8 out of 24 hours, resembling the standard human work schedule. The same number of minutes is spent working: in the second schedule, there are 3 times as much workers active during the 8 working hours during the 24 hours of the first schedule.

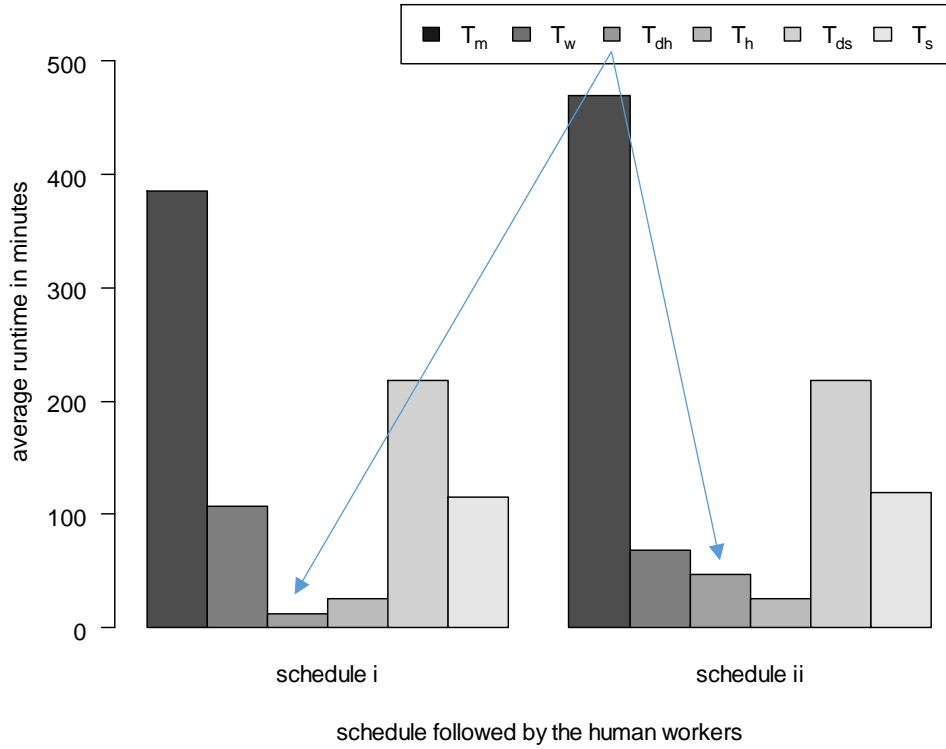


Figure 7.13: Average runtime when comparing the different worker schedules

Figure 7.13 shows the difference in average runtime between schedule i and schedule ii. The arrows show the difference in human delay, T_{dh} . This delay is about 4 times higher when using schedule ii, the 8-hour work day, when compared to schedule i, the 24-hour work shifts. This difference is due to the fact that the arrival of workflows is not bound to the 8-hour workday in our workloads. Workflows arriving towards or just after the end of the workday, can only get a human to finish the ‘human in the loop’ component 16 hours later, increasing the T_{dh} .

When executing workflows with a ‘human in the loop’ component that can arrive any time of the day unrelated to each other, 24-hour work shifts are much more effective than 8-hour work days, due to the direct response and reduced T_{dh} .

Results for Experiment H

The goal of experiment h, as shown in section 7.1, is to determine the stability of the system under test. For this experiment, each run has the same input parameters: The Learning provisioning policy with scaling threshold set to 15 and 9 humans working in 24-hour shifts.

Figure 7.14 and Figure 7.15 show the absolute and relative runtimes for the 10 runs of experiment h, respectively. There are differences between the runs, for example the human delay time in run 3 is longer than the other runs, this is due to the fact that there is a randomness in which tasks are picked up first, the interval in which the worker components check for new work, and, the exact duration of tasks executed by the workers.

These differences are a lot less than the differences in for example, experiments e and f, which leads us to believe there is no reason to examine this further.

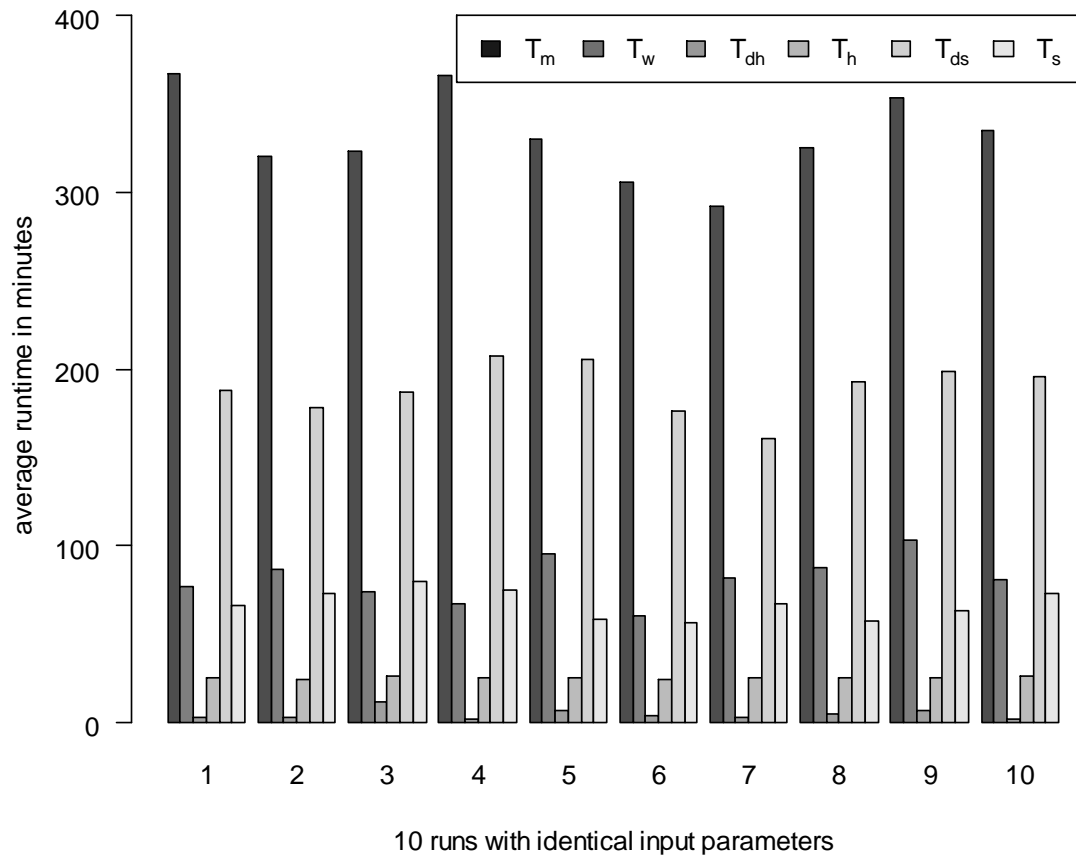


Figure 7.14: Average runtimes of the 10 runs with identical input parameters

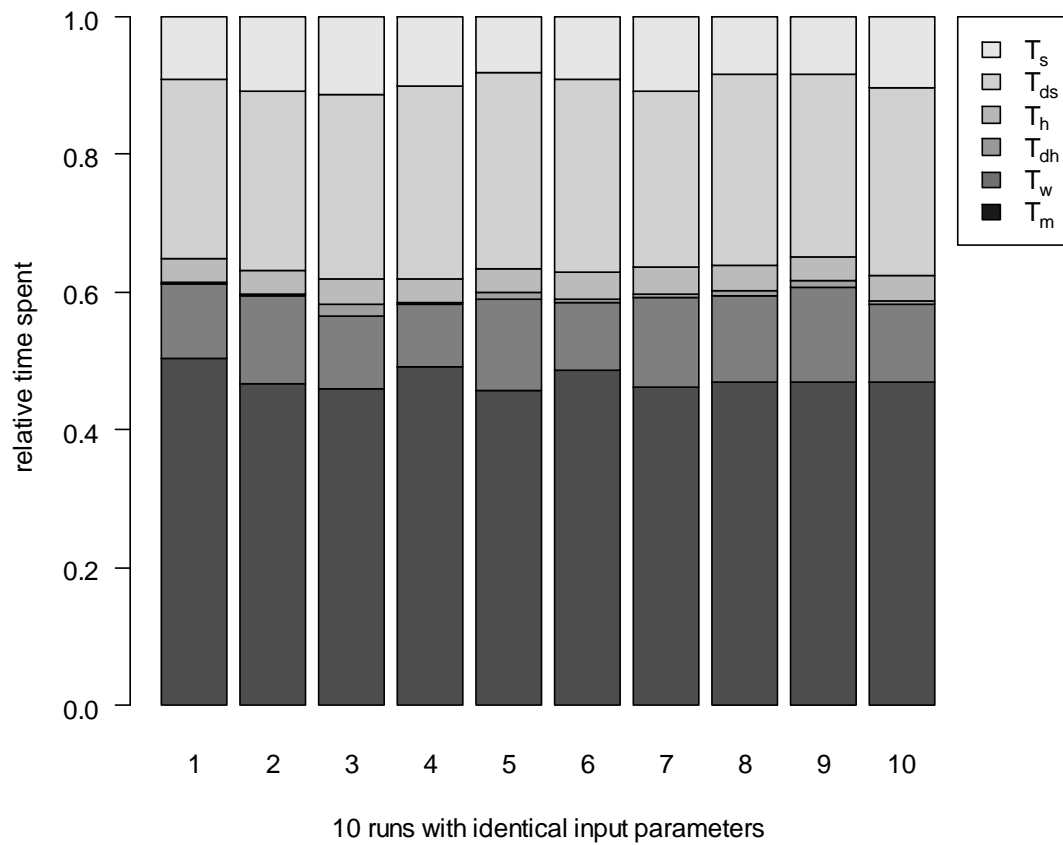


Figure 7.15: Relative runtimes of the 10 runs with identical input parameters

Recommendations

Giving a single recommendation after such diverse experiments is no easy task. Especially the added element of humans in the loop makes the BPaaS system interesting, this is why we will be giving several recommendations:

- **Scheduling policy: Learning.**
The learning policy, while only one variant has been experimented with, gives great results. This policy is promising and gives a faster response, as it uses BPaaS-related metrics. This will probably always be faster and more accurate than using system metrics like *load* or *disk usage*. When using the Learning policy, a low value for the scaling threshold is beneficial. For different workloads, these values can vary.
- **Human in the loop: 24-hour work shifts are slightly better than 8-hour workdays.**
Working in shifts, is the better choice when the arrival time of the workloads is unrelated to working hours. This could very well be different with workloads that *are* related to the standard 8-hour workday. We did not take extra costs for working in shifts into account.
- **Threads per node: 2.**
This is the most definite recommendation we can make. Our earlier prototypes worked with 4, and at times 5, threads per node. This gave us some, at the time, confusing results. After experiment A, it is clear that with our test-VM (1 vCPU, 1.75GB memory, 300 IOPs max) utilizing more than 3 threads is detrimental to the results, with 2 threads per node performing best.

7.4. Summary

In this Chapter, we validated the BPaaS reference architecture implementation: Elastic BPM. We first presented the research goals and corresponding experiments, the research methodology, and the metrics we measure during these experiments.

The infrastructure required to run the experiments and the approach for testing ‘humans in the loop’ is explained, after which the results from running the experiments and the ensuing recommendations are presented.

The results from our experiments and the recommendations presented are a good starting point for any BPaaS system. In Chapter 8, we discuss future work, which also includes a recommendation on scheduling policies.

8. Conclusion, recommendations, and future work

Each business has at least a few business processes [1], ranging from the automation of mundane tasks, to the regulation of large financial markets. These business processes are traditionally executed by expensive monolithic systems hosted on costly infrastructure without any ability to scale, leaving small NGOs, start-ups and schools without good options for executing business processes and large enterprises without good options for scaling their business. [2], [3] For this reason, we have defined the main research question of this thesis:

How to develop a cloud based business process service, such that companies requiring business process management can make use of this service on a pay per use basis, ensuring useful trade-offs in the cost-scalability space?

In Chapter 2, we have examined the history and context for this research question, examining the technical- and business aspects of virtualization, cloud computing, and business processes. Each following chapter of this thesis has added a new contribution to the BPaaS field of research.

- RC1:** We have addressed a comprehensive definition of a cloud based business process service in Chapter 3, where we have also introduced 5 use cases for BPaaS, and compared our definition with other definitions found in existing literature. The key difference we found with existing literature is the use of humans in the loop. Our definition includes this aspect and is key to our use cases.
- RC2:** Elastic-BPM, a first reference architecture for business process as a service, with key features being: (1) the user dashboard and process manager, responsible for the creation, execution, and monitoring of business processes, (2) the virtual infrastructure making use of small, preemptable components, to enable the pay-per-use business model, and (3) the scheduler, responsible for the trade-offs in the cost-scalability space has been described in Chapter **Error! Reference source not found..** We made the requirements for a BPaaS solution explicit and compared Elastic-BPM with alternatives found in literature.
- RC3:** Provisioning policies that provide useful trade-offs in the cost-scalability space for business processes, have been introduced in Chapter **Error! Reference source not found..** We have described policies that address the business process problem space, starting from a baseline static policy, including a naïve on-demand policy, and, introduced new, domain-specific, policies for BPaaS applications. We have explained the results of the experimentation with these policies in Chapter **Error! Reference source not found..**
- RC4:** A prototype implementation of Elastic-BPM, has been documented in detail in Chapter **Error! Reference source not found..** Our prototype implementation of the reference architecture Elastic-BPM, is a starting point for research on BPaaS. This research, including experiments run on the Microsoft Azure cloud, using a synthetic but diverse workload based on the use cases defined in Chapter 3, and the results of this research have been presented in Chapter **Error! Reference source not found..**

The remaining two sections of this final chapter will bring recommendations for building BPaaS applications, based on lessons learned while implementing the prototype for Elastic-BPM (section 8.1), and list possibilities for future work and research options (section 8.2).

8.1. Recommendation for Building BPaaS Applications

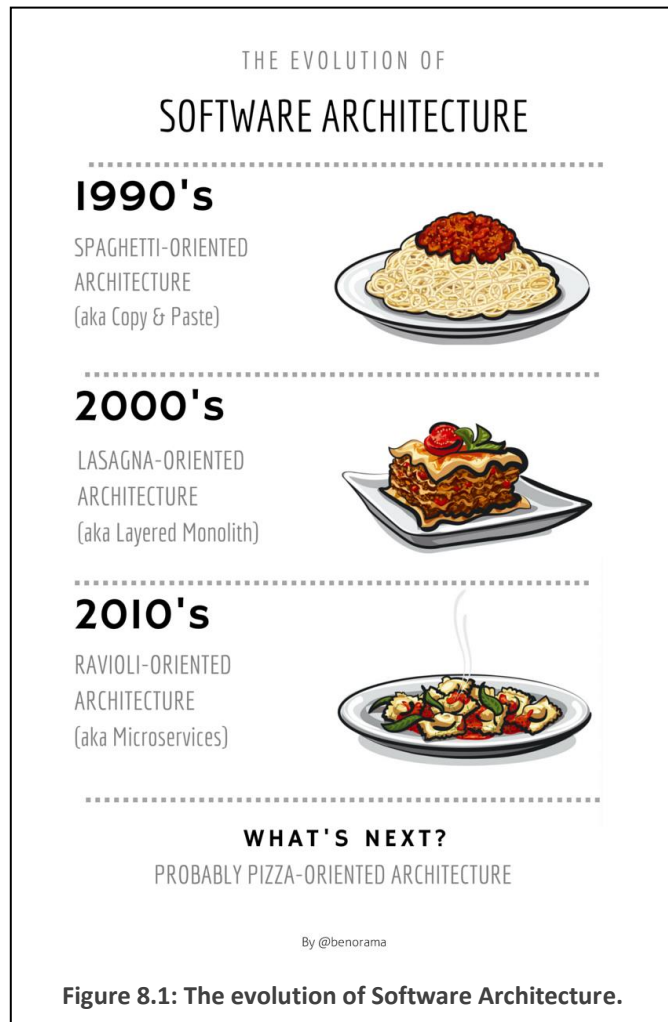
How should a new BPaaS application be built? The most logical choice for any 'as-a-Service' application will be to make use of the possibilities current cloud providers offer.

The choice between a Microservice architecture and a more traditional layered application, as discussed in section 0, is not as complicated as it used to be. Most of the pitfalls with Microservices have been described in great length and the frameworks for facilitating the creation and deployment of such systems are maturing at a rapid pace.

We recommend that any new (greenfieldⁱ) solution is built using the Microservices approach, combined with the Agile way of working. Choose a platform that fits your needs and budget, and start small.

Within businesses, however, completely new solutions are rare. Most solutions are either implemented using older, known and proven techniques, or are required to fit the already in-place languages, practices, and, architecture. When extending or improving an existing solution, we argue to take lessons from the Microservices and Agile advances in technology, and transform the in-place architecture to fit the modular, loosely-coupled Microservices architecture.

Start small and continuously improve.



What comes next?

Nobody knows (there might be a hint in Figure 8.1) – but be sure to watch the 'Serverless' space.ⁱⁱ [75], [76]

ⁱ Introduction to Greenfield Projects: en.wikipedia.org/wiki/Greenfield_project

ⁱⁱ Martin Fowler's take on Serverless: martinfowler.com/articles/serverless.html

8.2.Future work

In this section we present future improvements on the functionality, research, scheduling, and architecture of BPaaS and Elastic-BPM.

Elastic-BPM Functionality & Research Possibilities

- **Introduce tasks which cannot be pre-empted.**
Not every task in Business Processes can be pre-empted. For example: making a purchase order should not be pre-empted after the money has been paid. If this task is re-started at a later time, the money will be paid twice or more times.
- **Implement re-flow of workflow execution.**
In practical usage of workflow tooling, some form of re-flow during the execution of the workflow is desired. For example, when a certain task needs to be executed again, or skipped, for a specific case, and in the event of error handling.
- **Implement a way to directly import BPMN 2.0 XML into the BPaaS system.**
The current BPaaS system accepts a simplistic version of the BPMN standard. For a working enterprise solution and more interoperability, a better import is needed.
- **Simulate the human component better.**
Humans take breaks, and have vary levels of productivity during the day. The simulation currently does not represent these details in full detail, which can affect the level of realism of the proposed solution. When this simulation is made more realistic, the result more accurately represents real-life performance.

Scheduling Algorithm Improvements

- **Implement scaling based on heuristics and feedback from earlier workloads.**
The current implementation is based on the number of tasks in the ‘todo’ state, not taking into account type of workflow or task. The outline for these changes have been described in section 5.3.3.

Technical and Architectural Improvements

- **Implement a BPaaS system using ‘Serverless Computing’ (also known as Functions as a Service).**
We have implemented a BPaaS system while making use of Containers as smallest units of computation. The ‘Serverless Computing’ paradigm allows for even smaller granularity of computing, making it possible to optimize for *cost per task* instead of *cost per workload*.
- **Implement a BPaaS system using a distributed ‘Blockchain’ peer-to-peer approach.**
Currently there is a centralized approach to the BPM architecture, with a cost aspect that is paid by a single user. There might be a case for a decentralized multi-actor approach where the actors work together to host the system and execute the processes in combination with micropayments.ⁱ

ⁱ Blockchain, Ethereum, and Bitcoin are examples of a new approach. [78], [79]

References

- [1] W. M. P. Van Der Aalst, M. La Rosa, and F. M. Santoro, "BPM use cases - Structuring the business process management discipline," *Bus. Inf. Syst. Eng.*, vol. 6, no. 5, pp. 309–310, 2014.
- [2] R. K. L. Ko, S. S. G. Lee, and E. Wah Lee, *Business process management (BPM) standards: a survey*, vol. 15, no. 5. 2009.
- [3] J. vom Brocke and M. Rosemann, Eds., *Handbook on Business Process Management 2*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.
- [4] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities," *Proc. - 10th IEEE Int. Conf. High Perform. Comput. Commun. HPCC 2008*, pp. 5–13, 2008.
- [5] N. Paternoster, C. Giardino, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software development in startup companies: A systematic mapping study," *Inf. Softw. Technol.*, vol. 56, pp. 1200–1218, 2014.
- [6] A. Iosup and D. Epema, "Grid Computing Workloads : Bags of Tasks , Workflows , Pilots , and Others," *Condor, The*, pp. 1–8, 2010.
- [7] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," *I-SPAN 2009 - 10th Int. Symp. Pervasive Syst. Algorithms, Networks*, pp. 4–16, 2009.
- [8] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 931–945, 2011.
- [9] S. Shen, K. Deng, A. Iosup, and D. Epema, "Scheduling jobs in the cloud using on-demand and reserved instances," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8097 LNCS, pp. 242–254, 2013.
- [10] M. A. Khoshkholghi, A. Abdullah, R. Latip, S. Subramaniam, and M. Othman, "Disaster Recovery in Cloud Computing: A Survey," *Comput. Inf. Sci.*, vol. 7, no. 4, p. 39, 2014.
- [11] T. Wood, E. Cecchet, K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani, "Disaster recovery as a cloud service: Economic benefits & deployment challenges," *2nd USENIX Work. Hot Top. Cloud Comput. Boston, MA*, pp. 1–7, 2010.
- [12] C. Zenon, M. Venkatesh, and A. Shahrzad, "Availability and Load Balancing in Cloud Computing," *Int. Conf. Comput. Softw. Model. IPCSIT vol.14 IACSIT Press. Singapore*, vol. 14, pp. 134–140, 2011.
- [13] R. Kaur and P. Luthra, "Load Balancing in Cloud Computing," pp. 375–381, 2014.
- [14] M. Mishra and A. Das, "Dynamic resource management using virtual machine migrations," *IEEE Commun. Mag.*, vol. 50, no. September, pp. 34–40, 2012.
- [15] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena, "Intelligent workload factoring for a hybrid cloud computing model," *Serv. 2009 - 5th 2009 World Congr. Serv.*, no. PART 1, pp. 701–708, 2009.
- [16] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-degree compared," *Grid Comput. Environ. Work. GCE 2008*, 2008.
- [17] M. Armbrust, A. Fox, R. Griffith, A. Joseph, and RH, "Above the clouds: A Berkeley view of cloud computing," *Univ. California, Berkeley, Tech. Rep. UCB*, pp. 07–013, 2009.
- [18] M. Armbrust et al., "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, p. 50, 2010.
- [19] T. Dörnermann, E. Juhnke, and B. Freisleben, "On-demand resource provisioning for BPEL workflows using amazon's elastic compute cloud," *2009 9th IEEE/ACM Int. Symp. Clust. Comput. Grid, CCGRID 2009*, no. 2, pp. 140–147, 2009.
- [20] S. P. Mirashe, "Cloud Computing," *Commun. ACM*, vol. 51, no. 7, p. 9, 2010.

- [21] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing - The business perspective," *Decis. Support Syst.*, vol. 51, no. 1, pp. 176–189, 2011.
- [22] K. Bakshi, "Considerations for cloud data centers: Framework, architecture and adoption," in *2011 Aerospace Conference*, 2011, pp. 1–7.
- [23] R. Buyya, J. Broberg, and A. Goscinski, *Cloud Computing: Principles and Paradigms*. 2011.
- [24] S. Bhardwaj, L. Jain, and S. Jain, "Cloud Computing : a Study of Infrastructure As a Service (IaaS)," *Int. J. Eng.*, vol. 2, no. 1, pp. 60–63, 2010.
- [25] G. Lawton, "Developing Software Online With Platform-as-a-Service Technology," *Computer (Long Beach, Calif.)*, vol. 41, no. 6, pp. 13–15, Jun. 2008.
- [26] M. Turner, D. Budgen, and P. Brereton, "Turning software into a service," *Computer (Long Beach, Calif.)*, vol. 36, no. 10, pp. 38–44, Oct. 2003.
- [27] M. P. Papazoglou, "Service -oriented computing: Concepts, characteristics and directions," in *Proceedings - 4th International Conference on Web Information Systems Engineering, WISE 2003*, 2003, pp. 3–12.
- [28] A. Benlian and T. Hess, "Opportunities and risks of software-as-a-service: Findings from a survey of IT executives," *Decis. Support Syst.*, vol. 52, no. 1, pp. 232–246, 2011.
- [29] F. Oliveira *et al.*, "Delivering software with agility and quality in a cloud environment," *IBM J. Res. Dev.*, vol. 60, no. 2–3, p. 10:1-10:11, Mar. 2016.
- [30] B. Arnold and S. A. Baset, "Building the IBM Containers cloud service," *IBM J. Res. Dev.*, vol. 60, no. 2, pp. 1–12, Mar. 2016.
- [31] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure DevOps," in *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, 2016, pp. 202–211.
- [32] B. Yu, Z. Guan, Y. Jiang, C. Qi, and S. Liu, "The Container-Cloud Architecture and Scene Perception in IoE Era," *Int. J. Grid Distrib. Comput.*, vol. 9, no. 5, pp. 157–174, 2016.
- [33] A. Zenzinov, "Automated Deployment of Virtualization-Based Research Models of Distributed Computer Systems," in *Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2013)*, 2013, pp. 128–132.
- [34] G. G. Claps, R. Berntsson Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," in *Information and Software Technology*, 2015, vol. 57, no. 1, pp. 21–31.
- [35] K. An, S. Shekhar, F. Caglar, A. Gokhale, and S. Sastry, "A cloud middleware for assuring performance and high availability of soft real-time applications," *J. Syst. Archit.*, vol. 60, no. 9, pp. 757–769, 2014.
- [36] E. M. Maximilien, A. Ranabahu, R. Engehausen, and L. Anderson, "IBM altocumulus: a cross-cloud middleware and platform," *Proceeding 24th ACM SIGPLAN Conf. companion Object oriented Program. Syst. Lang. Appl.*, p. 805, 2009.
- [37] D. De Oliveira, E. Ogasawara, F. Baião, and M. Mattoso, "SciCumulus: A lightweigh cloud middleware to explore many task computing paradigm in scientific workflows," *Proc. - 2010 IEEE 3rd Int. Conf. Cloud Comput. CLOUD 2010*, pp. 378–385, 2010.
- [38] Consortium for Computing Sciences in Colleges. and ACM Digital Library., *Journal of computing sciences in colleges.*, vol. 25, no. 3. Consortium for Computing Sciences in Colleges, 2002.
- [39] J. O. Benson, J. J. Prevost, and P. Rad, "Survey of automated software deployment for computational and engineering research," in *10th Annual International Systems Conference, SysCon 2016 - Proceedings*, 2016, pp. 1–6.
- [40] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, 2014.

- [41] R. Morabito, J. Kjallman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393.
- [42] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," *Technology*, vol. 25482, pp. 171–172, Mar. 2014.
- [43] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer (Long. Beach. Calif.)*, vol. 40, no. 11, 2007.
- [44] D. Frith, "Cloud Deployments: Is this the End of N-Tier Architectures?," in *ISSE 2015: Highlights of the Information Security Solutions Europe 2015 Conference*, 2015, pp. 74–86.
- [45] L. Bass, I. M. Weber, and L. Zhu, *DevOps : a software architect's perspective*. .
- [46] G. Steinacker, "Scaling with Microservices and Vertical Decomposition," 2014.
- [47] W. Hasselbring and W. Hasselbring, "Microservices for Scalability," 2016.
- [48] S. White and R. Shapiro, *BPMN 2.0 Handbook*. Florida: Lighthouse Point, 2011.
- [49] T. Barton and C. Seel, "Business Process as a Service – Status and Architecture," pp. 145–158, 2013.
- [50] R. Accorsi, "Business Process as a Service: Chances for Remote Auditing," in *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, 2011, pp. 398–403.
- [51] T. Lynn *et al.*, "Towards a Framework for Defining and Categorizing Business Process-as-a-Service (BPaaS)," no. June, 2014.
- [52] A. Ilyushkin *et al.*, "An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows," *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. - ICPE '17*, vol. 1691, pp. 75–86, 2017.
- [53] D. Karastoyanova, T. Van Lessen, F. Leymann, and Z. Ma, "A Reference Architecture for Semantic Business Process Management Systems 1," *In Pract.*, vol. 27, no. 1, pp. 1727–1738, 1998.
- [54] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design."
- [55] M. Armbrust *et al.*, "Scaling Spark in the Real World: Performance and Usability."
- [56] F. Färber and N. May, "The SAP HANA Database--An Architecture Overview.," *IEEE Data ...*, pp. 1–6, 2012.
- [57] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges," *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7–18, 2010.
- [58] L. M. Vaquero, L. Roderio-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, 2011.
- [59] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," *Proc. - 12th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGrid 2012*, pp. 612–619, 2012.
- [60] C. Lu, J. Stankovic, S. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Syst.*, 2002.
- [61] J. A. Stankovic *et al.*, "Feedback control scheduling in distributed real-time systems," *Proc. 22nd IEEE Realt. Syst. Symp. RTSS 2001 Cat No01PR1420*, pp. 59–70, 2001.
- [62] R. C. Martin and M. Martin, *Agile principles, patterns, and practices in C#*. Prentice Hall, 2007.
- [63] K. Schwaber, *Agile Software Development with Scrum*. 2004.
- [64] A. Gunawan and K. M. Ng, "Solving the teacher assignment problem by two metaheuristics," *Int. J. Inf. Manag. Sci.*, vol. 22, no. 1, pp. 73–86, 2011.
- [65] K. Beck, *Test-driven development: by example*. 2003.
- [66] D. . Janzen and H. . Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *Computer (Long. Beach. Calif.)*, vol. 38, no. 9, pp. 43–50, 2005.
- [67] B. George, L. Williams, and W. L. George B., "An initial investigation of test driven development in

- industry,” *Proc. ACM Symp. Appl. Comput.*, pp. 1135–1139, 2003.
- [68] S. Berner, R. Weber, and R. Keller, “Observations and lessons learned from automated testing,” *Proceedings. 27th Int. Conf. Softw. Eng. 2005. ICSE 2005.*, pp. 571–579, 2005.
- [69] J. L. Carlson, “Redis in Action,” *Media.johnwiley.com.au*, 2013.
- [70] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” *Building*, vol. 54, p. 162, 2000.
- [71] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. 2015.
- [72] Y. Kouki, F. A. De Oliveira, S. Dupont, and T. Ledoux, “A language support for cloud elasticity management,” in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, 2014, pp. 206–215.
- [73] Ihaka and R. Gentleman, “R: {A} Language for Data Analysis and Graphics,” *J. Comput. Graph. Stat.*, vol. 5, pp. 299–314, 1996.
- [74] A. Maraschini, P. Massonet, H. Muñoz, and I. D. Telefónica, “When One Cloud Is Not Enough,” *Computer (Long. Beach. Calif.)*, pp. 44–51, 2011.
- [75] I. Baldini *et al.*, “Serverless Computing: Current Trends and Open Problems,” *arXiv Prepr. arXiv*, pp. 1–20, 2017.
- [76] E. Van Eyk *et al.*, “The SPEC Cloud Group’s Research Vision on FaaS and Serverless Architectures,” in *Proceedings of the 2nd Workshop on Second International Workshop on Serverless Computing (WoSC at Middleware 2017), Las Vegas, NV, USA, Dec 12, 2017*, 2017, p. article 1.
- [77] A. H. Maslow, “The Psychology of Science.” 1966.
- [78] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” *Www.Bitcoin.Org*, p. 9, 2008.
- [79] G. Wood, “Ethereum: a secure decentralised generalised transaction ledger,” *Ethereum Proj. Yellow Pap.*, pp. 1–32, 2014.