Integrated Timed Architectural Modeling/Execution Language

Lorenzo Bacchiani[1](✉), Mario Bravetti[1], Saverio Giallorenzo[1,2], Jacopo Mauro[3], and Gianluigi Zavattaro[1,2]

[1] Università di Bologna, Bologna, Italy
lorenzo.bacchiani2@unibo.it
[2] Focus Team, INRIA, Sophia Antipolis, France
[3] University of Southern Denmark, Odense, Denmark

Abstract. We discuss an integrated approach for the design, specification, automatic deployment and simulation of microservice-based applications based on the ABS language. In particular, the integration of architectural modeling inspired by TOSCA (component types/port dependencies/architectural invariants) into the ABS language (static and dynamic aspects of ABS, including component properties, e.g., speed, and their use in timed/probabilistic simulations) via dedicated annotations. This is realized by the integration of the ABS toolchain with a dedicated tool, called Timed SmartDepl. Such a tool, at ABS code compile time, solves (starting from the provided architectural specification) the optimal deployment problem and produces ABS deployment orchestrations to be used in the context of timed simulations. Moreover, the potentialities and the expressive power of this approach are confirmed by further integration with external tools, e.g.: the Zephyrus tool, used by Timed SmartDepl to solve the optimal deployment problem via constraint solving, and a machine learning-based predictive module, that generates in advance data to be used in a timed ABS simulation exploiting such predicted data (e.g., simulating the usage, during the day, of predicted data generated during the preceding night).

1 Introduction

Inspired by service-oriented computing, microservices structure software system as highly modular and scalable compositions of fine-grained and loosely-coupled services [23]. These features support modern software engineering practices, like continuous delivery/deployment [27] and autoscaling [8]. A significant problem in these practices is the automation of the deployment process of non-trivial microservice systems: cost-optimal distribution of components over the available Virtual Machines (VMs) and dynamic reconfiguration. Indeed, the ability to modify the system architecture during execution is a fundamental property to cope with adaptation needs, e.g., fluctuating peaks of user requests.

Although these practices are already beneficial, they can be further improved by exploiting the interdependencies within an architecture (interface functional

F. de Boer et al. (Eds.): *Active Object Languages: Current Research Trends*, LNCS 14360, pp. 169–198, 2024.
https://doi.org/10.1007/978-3-031-51060-1_7

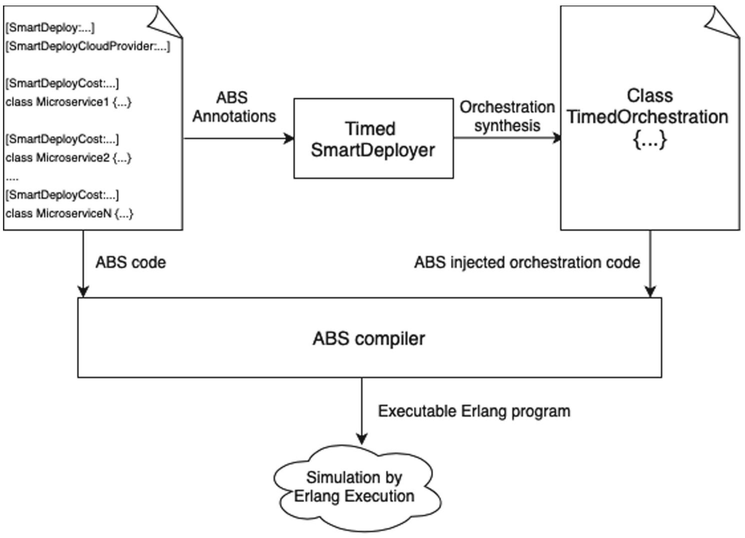

[SmartDeploy:...]
[SmartDeployCloudProvider:...]
[SmartDeployCost:...]
class Microservice1 {...}
[SmartDeployCost:...]
class Microservice2 {...}
....
[SmartDeployCost:...]
class MicroserviceN {...}
ABS Annotations
Timed SmartDeployer
Orchestration synthesis
Class TimedOrchestration {...}
ABS code
ABS injected orchestration code
ABS compiler
Executable Erlang program
Simulation by Erlang Execution

Fig. 1. Integrated timed architectural modeling/execution language toolchain.

dependences), instead of focusing on the single microservice. For instance, in the case of time-varying workload peaks w.r.t. traditional local scaling techniques [26], architecture-level dynamic deployment orchestration can avoid "domino" effects of unstructured scaling, i.e., single services scaling one after the other (cascading slowdowns) due to local workload monitoring.

In this paper, we thoroughly present the *integrated timed architectural modeling/execution language* introduced in [10]. The combination of modeling and execution capabilities makes it possible, in the context of a single language, to both (*i*) declaratively describe the architecture, its invariants, and the allowed reconfigurations and (*ii*) simulate system execution. Such an integrated language relies on an extension of the actor-based timed object-oriented Abstract Behavioral Specification (ABS) language [3]. In particular, it crucially exploits the twofold nature of ABS, which is both a process algebra (with probabilistic/-timed formal semantics) and a programming language (compiled and executed, e.g., with the Erlang backend), allowing for timed simulations. As can be seen in Fig. 1, we extend the ABS language with *Timed SmartDeployer* tool [10] annotations, which make it possible to express: *architectural properties* of the modeled distributed system (global architectural invariants and allowed reconfigurations), of its VMs (their characteristics and the resource they provide) and of its software components/services (their resource/functional requirements). Timed SmartDeployer, at compile-time, checks the satisfiability of such annotations accounting for the desired target configuration requirements, modeled using the *Declarative Requirement Language* (DRL) [20], and architectural invariants. Once the annotations have been validated, it synthesizes the *deployment* orchestrations that build the system architecture and each of its specified reconfigurations (via

DRL). Simmetrically, it also generates the *undeployment* orchestrations to undo such reconfigurations. More precisely, Timed SmartDeployer uses ABS itself as an orchestration language and makes (un)deployment ABS code available via methods with conventional names. In this way, such methods can be invoked by the ABS code of services, thus simulating run-time adaptation. Technically, such (un)deployment orchestrations are *timed (un)deployment orchestrations*, which also manage time aspects of the simulation, e.g., dynamically adjusting VM speeds, based on actually used cpu cores, and setting VM startup times. Therefore, Timed SmartDeployer integrates architectural annotations and timed ABS, used as an execution language.

The fact that, besides combining them in a single language, we also integrate (via orchestration generation) modeling and execution capabilities, makes it possible to anticipate at design level performance-related issues. This fosters an approach where the analysis of the consequences of deployment decisions are available early on. Timed SmartDeployer checks (at compile-time) the synthesizability of deployment orchestrations that, at run-time, will ensure the system to be always capable of reaching the desired reconfiguration (specified via DRL). For example, in the case of time-varying workload such desired reconfigurations would aim at globally incrementing the computational power via service replication. In this way, we would have the guarantee that the system is always capable of adapting to positive/negative peaks of user requests, respecting the imposed Quality of Service. On the contrary, run-time deployment decisions, if left to loosely-coupled reactive scaling policies, could lead to a chaotic behavior.

Timed SmartDeployer has to solve the problem of synthesizing timed deployment orchestrations starting from a declarative description of desired reconfiguration requirements. Such a problem, called *optimal deployment problem*, has been proved to be algorithmically treatable for microservices only [16,17]. Timed SmartDeployer provides an interface with ABS, reading ABS annotations with DRL declarations and injecting code of synthesized (un)deployment orchestrations into the initial annotated ABS program. To do this, it relies on a pluggable external solver which outputs the synthesized architectural configuration (cost-optimal distribution of components over the available VMs), which is, then, translated by Timed SmartDeployer into (un)deployment orchestrations expressed as timed ABS code. Notice that, being the solver pluggable, Zephyrus2 can be replaced with any other (not necessarily constraint-based) solver, which takes as input a DRL declaration and produces an architectural configuration.

Concerning the simulation of a modeled microservice system, executable ABS code is based on a set of hard-coded data (ABS array), which is divided into two parts: the actual and predicted workload for the simulated time period. Concerning the predicted workload, such data is generated at compile-time using a pluggable predictive module. Specifically, we make use of a machine learning predictive-based module implementing a neural network, which generates the workload data performing inference on a previously trained network. The idea is that the simulation represents system execution during the daytime and the neural network is trained during the preceding night. Notice that, being the predictive module pluggable, such a machine learning-based one can be replaced with any other module which produces predicted workload data.

Finally, we show our modeling execution language to be capable of expressing architecture-level adaptable systems. In particular, we consider, as a running example, a realistic microservice application, i.e., the Email Message Analysis Pipeline taken from Iron.io [24]. In such an application scenario, we use, as a reconfiguration requirement, some given increment or decrement of the system Maximum Computational Load (MCL), i.e., the maximum supported frequency for inbound requests (workload). Such global reconfigurations are used, in the context of an algorithm for architecture-level run-time adaptation [10] (also referred to as global scaling algorithm) to reach any target MCL (target workload), which overcomes the shortcomings of the traditional local scaling approach [26].

As we show in [10], the idea is that by monitoring at run-time the inbound workload, our algorithm causes the system to be always in the reachable configuration that better fits such workload (and that has the minimum number of deployed microservice instances). As a matter of fact, it is advantageous (see [9]) to consider as a target workload for the algorithm not merely the monitored one, but also the predicted workload (generated by the predictive module). Thus, we devised a run-time technique, based on past observed differences (where the most recent ones are given the highest weight) between monitored and predicted workload, to combine them into a single target workload.

Concerning the Email Message Analysis Pipeline itself, its model is built by considering static aspects of the architecture (annotations) and ABS code modeling the behavior of services. We simulate system execution using inbound traffic inspired to the real Enron dataset in [28], representing the frequency of emails entering the system. In order to show the effectiveness of our global scaling algorithm and show the advantages of using a predictive module and a technique to mix forecasted data with monitored ones, we run comparison experiments to show its advantages w.r.t. other approaches. The obtained code fully exploits the expressive power of ABS, e.g., using both its timed and probabilistic features. [1]

The paper is structured as follows. In Sect. 2, we briefly introduce our approach to the automatated deployment of microservice applications and we present the Email Pipeline Processing system that we use as a running example. In Sect. 3, we describe the Architectural Modeling/Execution Language, including Timed SmartDeployer and how we model service MCL. In Sect. 4, we present how external tools, i.e., Zephyrus2 and our machine learning based predictive module, can be integrated with this language. In Sect. 5, we test the expressive power of the Architectural Modeling/Execution Language, showing the implementation of the global scaling. Finally, in Sect. 6, we conclude the paper and discuss related work.

[1] Complexity of our ABS process algebraic models is also witnessed by the fact that they led us to discover an error in the Erlang backend: it caused interferences in time evolution between unrelated VMs (it was solved thanks to our code).

2 Microservices Deployment and Running Example

We now introduce our approach to the automatated deployment of microservice applications and illustrate it with our running example, the Email Message Analysis Pipeline.

2.1 Automated Deployment of Microservices

In [16,17], Bravetti et al. formalize component-based software systems and the problem of their automated deployment as the synthesis of deployment orchestrations (which allocate instances of software components on VMs) to reach a given target system configuration. In particular, the deployment life-cycle of each component type is formalized as a finite-state automaton, whose states denote a deployment stage. Each state corresponds to a set of *provided ports* (operations exposed by a component that other components can use) and a set of *required ports* (operations of other components needed by a component to work at that deployment stage). More specifically, Bravetti et al. [16,17] consider the case of microservices, components whose deployment life cycle consists of two phases: (*i*) creation, which entails the *mandatory* establishment of initial connections, via so-called *strongly required ports*, with other available microservices, and (*ii*) binding/unbinding, which corresponds to the establishment of *optional* connections, specified as so-called *weakly required ports*, to other available microservices. The two phases make it possible to manage circular dependencies among microservices.

The notions of strongly and weakly required ports are present also in state-of-the-art deployment technologies like Docker Compose [22], which is a language for the definition of multi-container deployments. In Docker Compose users specify different relationships among containers using, e.g., the depends_on (resp. external_links) relations. Then, these relations impose (resp. do not impose) a specific startup order among the containers, similar to how the combination of strong (resp. weak) dependencies induce an ordering in the orchestration of microservices deployment.

In addition, Bravetti et al. [16,17] consider resource/cost-aware deployments by modeling also memory and computational resources—i.e., the number of virtual CPU cores (vCores in Azure), sometimes simply called virtual CPUs as in Amazon EC2 and Kubernetes [26]. In particular, the authors enrich both microservice specifications and VM descriptions with the number of resources they, respectively, need and supply.

A microservice *deployment orchestration* is a program in an *orchestration language* that includes primitives for (*i*) creating/removing a certain microservice together with its strongly required bindings and (*ii*) adding/removing weak-required bindings between some created microservices. Given an initial microservice system, a set of available VMs, and a new target system configuration (corresponding to the set of microservices to be deployed), the *optimal deployment problem* is the problem of finding the deployment orchestration that (*a*) satisfies core and memory requirements, (*b*) leads to a new system configuration where

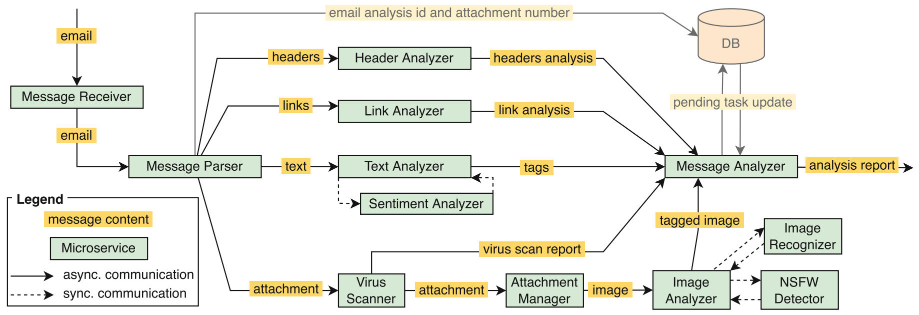


Fig. 2. Microservice Architecture of the Email Message Analysis Pipeline.

the target microservices are deployed, and (c) chooses the solution that optimizes resource usage, if more than one is available. As a typical example of an objective function to optimize, the reader can consider *cost minimization*, i.e., select among all possible deployment orchestrations the one which minimizes the sum of the cost-per-hour of the virtual machines used for microservice deployment.

While Di Cosmo et al. [18] proved that allowing components to have arbitrary deployment life-cycles makes the optimal deployment problem undecidable, Bravetti et al. showed that the latter becomes decidable when considering the simplified life-cycle of microservices described above, consisting of the two *creation* and *binding/unbinding* phases [16,17]. In particular, the authors presented a constraint-solving algorithm whose result is the new system configuration, i.e., the microservices to be deployed, their distribution over the VMs, and the bindings to be established among their strong/weak required and provided ports.

2.2 The Email Message Analysis Pipeline

In Fig. 2 (similar to that in [16,17]) we show a representation of the Email Message Analysis Pipeline [24]. The architecture includes 12 types of microservices, each equipped with its dedicated load balancer. Each load balancer distributes inbound requests among the set of microservice instances, whose number can change at runtime. We can logically partition our exemplary microservice application into four pipelines, each dedicated to the analysis or different parts of an email, namely its headers, links, text, and attachments (we detail each pipeline in Sect. 2.3). Messages enter the system through the *MessageReceiver*, which forwards them to the *MessageParser*. This microservice, in turn, extracts data from the email and routes them to the proper pipeline. Once each email component has been processed asynchronously (each taking its specific processing time), the *MessageAnalyzer* aggregates the outputs of each pipeline corresponding to the same email, and it produces a single analysis report for that email.

Before illustrating, in the next section, how one can apply to this example our approach for the automated deployment and scaling of microservice applications (cf. Sect. 2.1), we briefly present our representation of cloud resources.

We consider virtual CPU cores both for machines (providing them) and for microservices (requiring them). In particular, here, we assume microservices to be deployed on Amazon EC2 VMs of type *large*, *xlarge*, *2xlarge*, and *4xlarge*, each respectively providing 2, 4, 8, and 16 virtual CPU cores (following the Azure vCore terminology), simply called vCPUs in Amazon EC2. Notice that we model computational resources supplied by VMs (and required by microservices) using *virtual* cores with some speed fixed by the Cloud provider. Providers commonly use this kind of abstraction to uncouple the underlying hardware from the specifics exposed to users. Moreover, this level of indirection lets providers maximise the use of physical processors by delegating to the runtime (the VM/OS) the mapping of virtual cores and the scheduling of instructions. Each microservice type has a *number of required virtual cores*. Assigning the required virtual cores to a given microservice so that it achieves some expected performance (e.g., an estimated throughput) is a problem orthogonal to the one we investigate in this paper. While in practice programmers/operators perform this assignment as guesswork informed by their experience (as we do in this example), techniques like instruction counting [13] and profiling [14] can help in providing principled estimations.

2.3 Scaling Microservices

One of the pre-requisites to configure the deployment of microservice architecture is that each microservice should be defined by having a strongly required port towards the microservices which follow it in the pipeline. For instance, the *MessageParser* strongly requires connections with the *HeaderAnalyzer*, *LinkAnalyzer*, *TextAnalyzer*, and *VirusScanner* microservices since these services follow it in the pipeline (cf. Fig. 2).

More precisely, the ports should not be directly connected to instances of such microservices, but to their corresponding load balancers. In turn, each load balancer has a weakly required port that must be connected to all the available instances of the corresponding microservice, so that the load balancer can forward requests among them. The reasons behind this choice is that by establishing strongly requires connections to a microservice proxy it is possible to deploy first for example the load balancer of the *HeaderAnalyzer* and then deploy the instances of *MessageParser*, which are installable since they can be immediately connected to the load balancer they strongly require. Finally, it is possible to establish the connection of the load balancers to their instances through the weakly required port.

An advantage of using strongly and weakly required ports is that it is possible to easily capture dynamic adaptation of the pipeline deployment. A new microservice instance can be easily added to react to an increase workload by creating it and immediately connecting it to the (strongly required) load balancer of the microservice following in the pipeline. Then, the load balancer of the instance added binds to the new instance via the weakly required port. The removal of microservice instances instead follows the opposite order. First, we remove the binding between the load balancer of the microservice instances that we want to remove and, second, we safely de-allocate the interested instances.

Another advantage following from the knowledge of the microservice dependencies is that we can automatically adapt the whole architecture to provide the needed resources. Imagine for example the scenario in which an increase workload will require three new instances of *MessageParser* and two new instances of *HeaderAnalyzer* to proper handle all the traffic. If autoscaling [8] is used, scaling out and in decisions are taken locally by every service. As a consequence, the two services will be scaled out in sequence: first the *MessageParser* that comes first in the pipeline (and therefore witness for first the effects of the increase of the traffic) and then the *HeaderAnalyzer* that will start to be invoked more often by the *MessageParser*. Luckily, as shown in Sect. 5, having a global knowledge of the microservice dependencies allows to exploit the information that more than one service can be scale out at once and therefore perform a global adaptation. In our scenario, both the *MessageParser* and the *HeaderAnalyzer* would be scaled out at the same time, thus allowing the avoidance of the domino effects typical of autoscaling strategies.

2.4 Microservice Maximum Computational Load

We now introduce an important property of microservices, which characterizes their throughput: Maximum Computational Load (MCL), i.e., the maximum number of requests that a *microservice instance* of that type can handle within a second. As we will see, it is important to consider such a property to assess the correctness w.r.t. time behaviour of our integrated timed architectural modeling/execution language.

More precisely, the MCL of a microservice is computed as follows:

$$\mathsf{MCL} = 1/(\frac{\mathsf{size}_{\mathsf{request}}}{\mathsf{data\_rate}} + \mathsf{pf})$$

where $\mathsf{size}_{\mathsf{request}}$ is the average request size of the microservice in MB. Moreover, $\mathsf{data\_rate}$ is the microservice rate in MB/sec for managing request data. We determine such a value, based on the number of microservice requested cores, from Nginx server data in [31] (considering Nginx servers with that number of vCPUs). Finally, pf is a penalty factor that expresses an additional amount of time that a microservice needs to manage its requests, e.g., the *ImageRecognizer*, which needs Machine Learning techniques to fulfill its tasks.

3 Architectural Modeling/Execution Language

3.1 Abstract Behavioral Specification Language

Abstract Behavioral Specification (ABS) [3] is an actor-based object-oriented specification language (a process algebra) offering algebraic user-defined data types, side effect-free functions and immutable data. Since ABS is not directly executable, its toolchain [4] contains several backends that compile algebraic models into an executable programming language, e.g., Erlang in the case of the Erlang backend, and execute it. ABS objects are organized into Concurrent Object Groups (COGs) representing software components or services. Objects belonging to different COGs communicate with each other using asynchronous method calls [15], expressed as *object!method(...)* instructions. Asynchronicity is realized by means of the future mechanism: asynchronous method calls return a future that can be used to wait for the result using the *await* statement [5]. *Timed ABS* [7] is an extension to the ABS core language that introduces a notion of *abstract time*. In particular, evolution of time in ABS is modeled by means of discrete time: during execution system time is expressed as the number of *time units* that have passed since system start. The modeler decides what a time unit represents for a specific application. Such a feature makes it possible to perform simulations analysing the time-related behavior of systems. Timed ABS has also *probabilistic* features that allow modelers to create uniform distributions, e.g., the average number of attachments per email in our case study.

To represent VMs (and simulate them, e.g., inside the Erlang backend) ABS introduces the notion of Deployment Component (DC) [6] as a *location* where a COG can be deployed. As VMs, ABS DCs are associated with several kinds of resources, expressed via a dedicated annotations. In particular virtual cpu speed is represented in ABS by the DC *speed*: it models the amount of *computational resource* per time unit a DC can supply to the hosted COGs. This resource is consumed by ABS instructions that are marked with the *Cost* tag, e.g., *[Cost: 30] instruction*. COG instructions tagged with a cost consume the hosting DC computational resource still available for the current time unit (the instruction above consumes 30 from the DC speed resource): if not enough computational resource is left in the current time unit, then the instruction terminates its execution in the next one.

Concerning our approach to automated microservice deployment, based on strong and weak dependencies, in ABS we represent microservice types as classes and microservice instances as objects, each executed in an independent COG. Moreover, we represent strong dependencies as mandatory parameters required by class constructors: such parameters contain the references to the objects corresponding to the microservices providing the strongly required ports. Weak required ports are expressed by means of specific methods that allow an existing object to receive the references to the objects providing them.

3.2 Timed SmartDeployer

Timed SmartDeployer is executed at ABS compile time: it statically solves the optimal deployment problem described at the end of Sect. 2.1, i.e., synthesis of deployment orchestrations that reach a given target system configuration. Timed SmartDeployer takes its input from dedicated ABS annotations, which are present in the compiled ABS program, and produces its output as ABS code—synthesized timed (un)deployment orchestration—which is added to the initial annotated ABS program.

Timed SmartDeployer ABS Annotations. The JSON based ABS annotations from which Timed SmartDeployer extracts its input are:

- *[SmartDeployCost : JSONstring]* class annotation. This annotation is bound to an ABS class representing a given microservice type. It describes the functional dependencies (provided and weak/strong required ports) and the resources (e.g., number of cores and amount of memory) a microservice needs.
- *[SmartDeployCloudProvider : JSONstring]* global annotation. It defines the properties (e.g., *Cores, Bandwidth, Memory, Speed, StartupTime*) and cost-per-hour of the DCs created in the synthesized orchestration execution.
- *[SmartDeploy : JSONstring]* global annotation. It describes the desired properties and constraints of the deployment orchestration, e.g.:
  - The `id` property, which sets the name for the class that is going to include the ABS code of the synthesized orchestration.
  - The `cloud_provider_DC_availability` property, which fixes the maximum number of VMs the orchestration can allocate.

  Some of these properties can have, as JSON value, a string whose content is a declarative specification (a formula of a logic language that is based on first-order logic), e.g.:
  - The `specification` property, which contains the declarative specification of the desired configuration in DRL. A value for this property, taken from our running example (orchestration with id `Scale2` in [1], see Sect. 5 for its description), can be:

    ```
    SentimentAnalyser = 3 and VirusScanner = 2 and
        AttachmentsManager = 1 and ImageAnalyser = 1 and
    NSFWDetector = 2 and ImageRecognizer = 2 and
        MessageAnalyser = 2
    ```

    meaning that 3 instances of the *SentimentAnalyser* must be (additionally) deployed, etc.
  - The `bind preferences` property, which is used to specify preferences about *weak* bindings among service instances (using the declarative language of [20]). A value for this property, taken from our running example (orchestration with id `BaseScale` in [1], see "**B**" configuration in Sect. 5), can be:

    ```
    forall ?x of type MessageReceiver in '.*' :
      forall ?y of type MessageReceiver_LoadBalancer in '.*'
          : ?x used by ?y
    ```

    meaning that each instance (variable `?x`) of the *Message Receiver* service has to be bound to each *MessageReceiver_LoadBalancer* service instance (variable `?y`). Given that there exists only 1 instance of the *MessageReceiver_LoadBalancer* service in the system, this just means that each instance of the *MessageReceiver* service has to be bound to its load balancer. More precisely, the `in` keyword is used to set the scope for the indicated service: services considerd are only those located inside the DCs whose names are declared after `in`. Such a declaration can be made with a regular expression like $'.*'$ (meaning any string), i.e., the service can be located in any running DC.

Synthesized Timed (Un)deployment Orchestration. Timed SmartDeployer produces as output the desired *timed (un)deployment orchestration*: a timed ABS program, injected in the initial annotated one, containing the set of *orchestration language* instructions (expressed as timed ABS code). The execution of the newly synthesized orchestration causes the system to reach a deployment configuration with the desired properties.

Internal Details. As detailed above, Timed SmartDeployer provides an interface with ABS, reading ABS annotations with DRL declarations and injecting the synthesized (un)deployment orchestrations code into the initial annotated program. To do this, it relies on a pluggable external solver, e.g., the Zephyrus2 constraint solver [2].

The external solver outputs the synthesized architectural *configuration* (cost-optimal distribution of components over the available VMs), which is, then, translated by Timed SmartDeployer into (un)deployment orchestrations expressed as timed ABS code. Such *timed* deployment orchestrations additionally encompass (w.r.t. untimed ones, see Sect. 2.1) dynamic management of overall DC *startup time* and *speed* (computational resources per time unit, see Sect. 3.1), based on the number of DC virtual cores that are actually used by some microservice after enacting the synthesized deployment sequence. As we will show, this allows us to correctly model time (microservice MCL). Timed SmartDeployer *dynamically* assigns a speed and a startup_time to each DC that is created during a deployment orchestration. Such timed properties of created DCs are evaluated, starting from the speed and startup_time annotations (see Sect. 3.2) in the original ABS code, as follows. The speed property is dynamically evaluated, during orchestration execution, taking into account the number of DC cores that are actually used: $\mathsf{speed} - \mathsf{speed\_per\_core} \cdot \mathsf{unused\_cores}$. Concerning startup_time, we dynamically set an overall startup time such that it is the *maximum* among those of the DCs created during a deployment orchestration. The above is realized by automatically synthesizing timed orchestrations, whose language additionally includes (w.r.t. untimed ones) two primitives *explicitly* managing time aspects

– One to decrement the speed of a DC: *decrementResources(...)* in ABS.
– One to set overall the startup time of created DCs: *duration(...)* in ABS.

3.3 Modeling Service MCL

We now show how Time SmartDeployer allows us to correctly simulate the service MCL we want to model (see Sect. 2.4), independently of the VM (DC) in which it is deployed. An example is considering, as we do in our case study, the ABS time unit to be 1/30 *sec* and setting VMs to supply 5 speed_per_core. According to the calculation we presented in Sect. 2.4, it turns out that the MCL of an actual implementation of the *ImageRecognizer* service is 91 requests per second. In the ABS code, to model service MCL, we make use of the *Cost* instruction tag (see Sect. 3.1). E.g., in the case of the *ImageRecognizer*, which requires 6 cores to be deployed, we obtain the MCL of 91 *req*/*s* as follows:

```
class ImageRecognizer() implements ImageRecognizerInterface {
  Int mcl = 91;
  String recognizeImage(ImageRecognizer_LoadBalancerInterface
      balancer){
    [Cost: 5 * 6 * 30 / mcl] balancer!removeMessage();
    Int category = random(9);
    return "Category Recognized: " + toString(category);
  }
}
```

where the method *recognizeImage*(...) is executed at each request.

Due to our SmartDeployer timed extension, the amount of VM speed used by *ImageRecognizer* is always $5 \cdot 6$ ($\mathsf{speed\_per\_core} \cdot \mathsf{cores\_required}$), independently of the VM in which it is deployed, i.e., *ImageRecognizer* can use up to $5 \cdot 6$ computational resources per time unit. The *Cost* tag above causes each request to consume $\mathsf{speed\_per\_core} \cdot \mathsf{cores\_required} \cdot 30/\mathsf{MCL}$ computational resources. Therefore, since $\mathsf{MCL}/30$ is the *ImageRecognizer* MCL expressed in requests per time unit, this realizes the desired (deployment independent) service MCL.

4 Integration with External Tools

In this section, we discuss external tools (w.r.t. ABS) that we have used in our work. First, we need a tool to solve the problem of synthesizing timed deployment orchestrations, starting from the deployment information contained in the ABS annotations. Second, given that we plan to use the executable semantics of ABS to simulate deployment and scaling policies for microservice systems that include also predictions of the incoming workload fluctuations, we also need a tool for workload prediction. Concerning the first tool, we have used the Zephyrus2 [2] solver based on constraint-solving technology, while for the latter we have adopted a well-established Machine Learning (ML) techniques. It is interesting to observe that, being such tools pluggable, Zephyrus2 and the ML predictive module could be replaced with any other (not necessarily constraint- or ML-based) tools.

4.1 The Zephyrus Deployment Engine

As described in Sect. 3.2, Timed SmartDeployer extracts, from ABS code, deployment information of different kinds: (*i*) *class annotations* that describe the requirements of objects which represent the resources and dependencies of the microservice instances modeled by such objects and (*ii*) *global annotations* that describes the available computing resources and the desired properties that the deployment should satisfy. Such annotations are processed by the deployment engine that automatically synthesizes a microservice architecture allocating the various microservices on available computing resources. This is done taking into account both local (e.g., single microservice dependencies) and the global (e.g., minimize the total number of allocated resources) constraints.

The deployment engine which is currently used in our Timed SmartDeployer prototype is Zephyrus2 [2]. Zephyrus2 is a tool for optimal deployment of software components over virtual machines that exploits SMT (Satisfiability Modulo Theories) and CP (Constraint Programming) technologies. More precisely, Zephyrus2 expects in input three different kinds of deployment information:

– a description of the components that can be deployed (which includes the consumed computing/memory resources as well as the functionalities required/provided from/to other components),
– a description of the virtual machines where the components can run (which includes the resources offered by the virtual machines as well as other information, like their cost), and
– the specific requirements on the component-based software architecture to be computed and deployed over the available virtual machines.

Notably, the last item could include also objective functions to be optimized, e.g., the request to minimize the total cost of the used virtual machines. Zephyrus2 then produces as output a description of the components to deploy, the allocation of such components over the available virtual machine, and the bindings among the components that reciprocally require/offer functionalities. The computed deployment satisfies the constraints and requirements specified in input.

Zephyrus2 computes its output as a solution to an optimization problem encoded in MiniZinc [29], a solver independent language for modeling constraint satisfaction and optimization problems. The interested reader can find in [2] details about how Zephyrus2 produces the MiniZinc specification of the deployment problem and how it exploits state-of-the-art tools to solve such problem. Here, we simply give an idea of how to translate deployment requirements into constraints on a couple of simple examples.

As a first example, we consider the allocation of memory to the components. Consider the constraint

$$\bigwedge_{v \in VM} \quad \sum_{C \in CompTypes} \texttt{inst}(C, v) \cdot C.mem \leq v.mem$$

where VM denotes the set of all the available virtual machines, $CompTypes$ the possibile component types, $\texttt{inst}(C, v)$ the number of instances of components of type C allocated on the virtual machine v, $C.mem$ the memory consumed by a component of type C, and $v.mem$ the memory available on the virtual machine v. This constraint enforces the requirement that it is not possible, on every virtual machine, to allocate to components strictly more than the available memory.

As an additional example, we consider how it is possible to require the deployment which minimizes the total cost. For all the virtual machines v, a new boolean variable $\texttt{used}(v)$ is introduced and bound to be true if at least a component is deployed on the v by the following constraints:

$$\bigwedge_{v \in VM} \Big(\sum_{C \in CompTypes} \texttt{inst}(C, v) > 0 \Big) \Leftrightarrow \texttt{used}(v)$$

Then to minimize the total cost is is possible to minimize the following objective function:

$$\min \sum_{v \in VM,\, \texttt{used}(v)} v.cost$$

where $v.cost$ is the cost of the virtual machine v.

4.2 ML-Based Predictive Module

When simulating a modeled microservice system using executable ABS code, we use a set of hard-coded data points in the form of an ABS array. While the most straightforward option is to run the simulation on actual traffic workload, our modular approach allows us to also integrate other components, such as *predictive modules*, which forecast traffic fluctuations, and *actuation modules*, which regulate how the logic for the architectural adaptation weights the different sources of information (e.g., the simulated traffic and its prediction).

Focusing on the role of prediction modules, we can distinguish between two types of information: the actual workload and the predicted workload. The latter is generated at compile-time using some pluggable predictive modules. For instance, one can implement the predictive module through neural networks, where the predictive module generates workload data by performing inference on the previously trained network. While this approach is apt for a simulation environment, it does not depart sensibly from real-world applications, e.g., where one can collect daytime information on the traffic and feed it to the neural model and obtain the forecast for the next day during the night.

Predicting the Traffic of the Email Message Analysis Pipeline. As an example, we describe how one can use data analytics to predict traffic fluctuations in our running example (cf. Sect. 2.2).

Dataset. The prediction module requires a datasat for training. Since the execution context of the Email Pipeline architecture is that of email correspondence, we draw our data from Enron corpus dataset [28]. This dataset has been made public by the Federal Energy Regulatory Commission during investigations concerning the Enron corporation (version of May 7th, 2015). The dataset contains 517,431 emails from 151 users, without attachments, distributed over a time window of about 10 years (1995–2005).

We processed the dataset to extract the attributes for predicting the number of incoming emails for a given time. We assume that time is discretized in one-hour intervals. For every email we extracted the *datetime* attribute, and then we summed the number of emails in the desired monitored time. Every email was associated with five new attributes: *month*, *day*, *weekday*, *hour*, and *counter* giving us a representation of the email flow in the system at a given hour.

Predictions. There are many techniques that one can apply to predict the traffic load. For our use case, as detailed in [9], the off-the-shelf Multi-Layer Perceptron is used. For the training, the dataset has been partitioned into a training set, a validation set, and a testing set—the latter, to estimate the error rate of the model. Specifically, a neural network with three fully-connected layers have been used, applying the Rectified Linear Unit (ReLU) nonlinear activation function to the output of each layer. Each level of the neural network compressed the input into a smaller representation. The first level reduced the input from 70 to 64 attributes, while the second level reduced it from 64 to 32 attributes. Finally, they linearly projected the 32 attributes into a single value that corresponds to the regression target. To compute the error rate, the Mean Squared Error (MSE) loss function is used while to optimize the network parameters the Adaptive Moment Estimation (Adam) has been used. The training process had a learning rate of 0.1 and an exponential decay scheduler with $\gamma = 0.9$.

5 Simulation of Architecture-Level Adaptable Systems

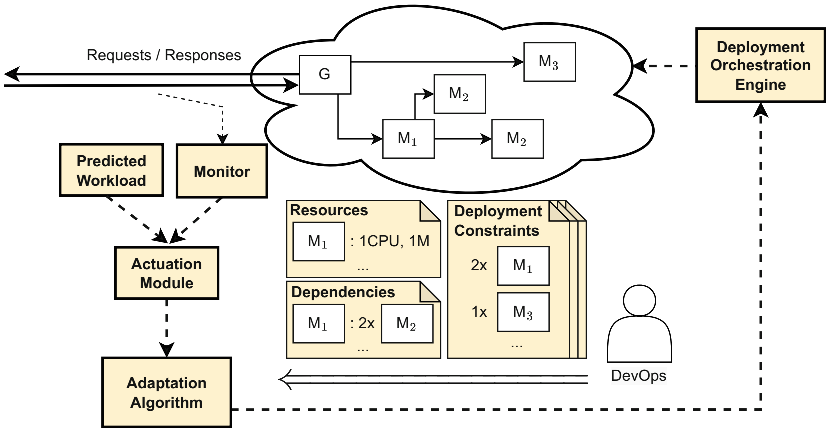


Fig. 3. Simulating proactive-reactive architecture-level system adaptation.

To test the expressive power of our modeling execution language, we simulate the platform depicted in Fig. 3. Such platform is made of two kinds of elements: the microservice system to be adapted (labelled G, $\mathsf{M}_1, \mathsf{M}_2$, M_3) and the element of the platform itself (depicted with orange boxes). Since the platform sees microservices as instance parameters, we abstract from their actual behaviour. We now describe each element of the platform. Before doing so, we highlight the three kinds of flows in Fig. 3: $\rightarrow$ showing the inbound workload reaching the microservice architecture; dashed-line arrows $\dashrightarrow$ regarding the runtime execution of an architecture-level adaptation process; the thick arrow $\Leftarrow$ indicating the compilation time of deployment orchestrations.

Deployment Orchestration Engine. This component receives a deployment orchestration and enacts all the operations it contains, e.g., (de)allocating VMs, microservice replications. It is a loosely-coupled component of the platform, taken from existing solutions (the only requirement is that it provides a programming interface for the application of deployment plans), such as Kubernetes. In our simulated environment, the deployment orchestration engine is represented by the Erlang backend, which is in charge of executing the whole simulation.

Adaptation Algorithm The Adaptation Algorithm implements an architecture-level adaptation algorithm that computes the deployment orchestrations to be applied in order to cope with inbound workload peaks. To do that, such module takes into account two inputs. The first one, represented by $\Leftarrow$, regards the deployment orchestrations statically computed by Timed SmartDeployer (see Sect. 3.2) by means of a constraint solver, e.g., Zephyrus2. These orchestrations are computed such that they satisfy the specifications given by the user (DevOps in Fig. 3), i.e., Resources, Dependencies and Deployment Constraints in Fig. 3, respectively included in the *SmartDeployCloudProvider, SmartDeployCost* and *SmartDeploy* annotations, see Sect. 3.2. The second input, represented by $\dashrightarrow$, regards the workload the system has to support, after the adaptation process. In this case, the Adaptation Algorithm acts as a service that other components call. Upon activation, the Adaptation Algorithm interacts with the Deployment Orchestration Engine to perform the scaling.

Monitor. The monitor tracks the traffic flowing on the architecture within a prefixed *time window* and checks the possible occurrence of a *workload deviation*, e.g., the difference between the monitored workload and the globally supported one, as we will see in the following sections. When such a condition occurs, the Monitor sends to the Actuation Module the amount of measured workload.

Predicted Workload. The Predicted Workload is computed by means of a predictive module external to the simulation. In our case, we perform predictions using the ML-based predictive module described in Sect. 4.2. Such workload is statically injected in the simulation exploiting a standard ABS data structure, i.e., arrays, and it is forwarded to the Actuation Module.

Actuation Module. The Actuation Module computes the amount of workload given as input, i.e., the *target workload*, to the Adaptation Algorithm. Depending on how such workload is computed, we distinguish 3 modalities: (*i*) *reactive* mode, if the target workload is the one measured by the monitor (this modality has no predicted workload); (*ii*) *proactive* mode, if the target workload is represented by the predictions in the Predicted Workload (this modality has no monitor); and (*iii*) *proactive-reactive* mode, if the target workload is computed as a combination of the signals coming from the Monitor and Predicted Workload, according to the mixing technique implemented in this module.

Concretely, we model the architecture platform and the scaling approaches via ABS, compiling it into a system of Erlang programs that run the simulation. Then, the simulation receives three kinds of inputs, which are statically defined within a simulation run: *deployment orchestrations* (generated by Timed SmartDeployer at compile-time, see Sect. 3.2), an *actual* and a *predicted workload* (generated by the Predictive Module, see Sect. 4.2) both hard-coded in the simulation in the form of arrays. We model a real-world request flow sent to the simulated microservice architecture via an ad-hoc generating service, which distributes requests as specified in the actual workload array. The simulation uses these inputs to evaluate the performance of a target microservices architecture.

5.1 Application to Global Scaling

In the following sections, we use the above presented architecture (see Fig. 3) to simulate the algorithm for global run-time adaptation that we introduced in [10]. Such an algorithm, which we could conceive and simulate thanks to our integrated timed architectural modeling/execution language, finds application in the context of cloud-computing platforms endowed with orchestration engines. The algorithm reaches, by performing global reconfigurations, a target *system Maximum Computational Load (MCL)*, i.e., the maximum supported frequency for inbound requests. The idea is that, by monitoring at run-time the inbound workload, our algorithm causes the system to be always in the reachable configuration that better fits such a workload (and that has the minimum number of deployed microservice instances). This is achieved by enacting global reconfigurations, which are targeted at guaranteeing a given increment/decrement of the system MCL.

In particular, in the next section, we introduce the concept of microservice Multiplicative Factor (MF), which is needed by the algorithm. We already observed that each microservice type is characterized by a MCL (see Sect. 2.4), i.e., the maximum number of requests that a *microservice instance* of that type can handle in a second. We additionally observe that each microservice type is also characterized by a MF, i.e., the mean number of requests that a single request (i.e., an email) entering the system generates for that *microservice type*.

In the remaining sections, we introduce all the building blocks needed to realize our global scaling approach. We start from the mathematical calculation of the global scaling reconfigurations incrementing/decrementing the system MCL. This is done by showing how system MCL can be computed by the MCL of single service instances, which, in turn, are mathematically calculated based on the microservice data rate (we use, e.g., real data in [31] for Nginx servers) and the role it plays in the application architecture (which determines its MF and the size of its requests for each incoming message). Such global reconfigurations are synthesized into deployment orchestrations by Timed SmartDeployer. We then show a technique to combine the monitored and predicted workload into a unique target workload, used in our proactive-reactive global scaling approach. We finally introduce the scaling algorithm showing its implementation via ABS code excerpts. We then simulate the introduced global scaling approach by applying it to our example (cf. Sect. 2.2) and present simulation results: a set of comparisons that, not only shows that our global scaling approach overcomes the limitations of the traditional local one, but also the extent of improvements brought by our predictive module (see in Sect. 4.2) and our technique for computing the target workload to a purely reactive global scaling approach.

5.2 Microservice Multiplicative Factor

The Multiplicative Factor (MF) of a microservice type is determined from the role it plays in the whole architecture, e.g., in the running example, by the email part it receives. As a consequence it is strictly related to the (average) structure of emails entering the system. In particular, we estimate an email to have: (i) a single header; (ii) a set of links (treated collectively as a single information, received by the *LinkAnalyser*); (iii) a single text body (received by the *TextAnalyser*), which is split, on average, into $\mathsf{N}_{\mathsf{blocks}} = 2.5$ text blocks (individually analysed by *SentimentAnalyser*); and (iv) on average $\mathsf{N}_{\mathsf{attachments}} = 2$ attachments (individually sent to the attachment sub-pipeline starting with the *VirusScanner*), each having average size of $\mathsf{size}_{\mathsf{attachment}} = 7\mathsf{MB}$ and containing a virus with probability $\mathsf{P}_{\mathsf{V}} = 0.25$ (which determines whether a virus scan report is sent to the *MessageAnalyser* or, in case of no virus, the attachment is forwarded to the *AttachmentManager*).

The average numbers above are estimated ones: the MF of microservices can be easily recomputed in case different numbers are considered. In particular, MFs are calculated as follows. Since emails have a single header, a set of links that are sent together and a single text body, the microservices that analyze these elements, i.e., *HeaderAnalyser, LinkAnalyser* and *TextAnalyser*, have $\mathsf{MF} = 1$. As text blocks and attachments are individually sent, each of them generates a request to the *SentimentAnalyser* and the *VirusScanner*, therefore they have $\mathsf{MF} = \mathsf{N}_{\mathsf{blocks}}$ and $\mathsf{MF} = \mathsf{N}_{\mathsf{attachments}}$ respectively. The microservices that follow the *VirusScanner* in the architecture, i.e., *AttachmentManager, ImageAnalyzer, ImageRecognizer* and *NSFWDetector* have a MF equal to the number of virus-free attachments, which can be computed as $\mathsf{MF} = \mathsf{N}_{\mathsf{attachments}} \cdot (1 - \mathsf{P}_{\mathsf{V}})$. Finally, the MF of the *MessageAnalyser* is the sum of the email parts (1 header, 1 set of links, 1 text body and $\mathsf{N}_{\mathsf{attachments}}$ attachments).

From a timing viewpoint, considering microservice type Maximum Computational Load (MCL) and MF is important because it allows us to calculate the minimum number of instances of that type needed to guarantee a given overall system MCL $\mathsf{sys\_MCL}$, i.e.[2]

$$\mathsf{N}_{\mathsf{instances}} = \left\lceil \frac{\mathsf{sys\_MCL} \cdot \mathsf{MF}}{\mathsf{MCL}} \right\rceil$$

Notice that, a microservice MF is also important in order to determine its request size $\mathsf{size}_{\mathsf{request}}$, which, in turn, as we showed in Sect. 2.4, is needed to calculate its MCL. More precisely, we compute microservice $\mathsf{size}_{\mathsf{request}}$ as follows. In our running example, for all microservices receiving attachments but the *MessageAnalyser* we have:

$$\mathsf{size}_{\mathsf{request}} = \mathsf{N}_{\mathsf{attach\_per\_req}} \cdot \mathsf{size}_{\mathsf{attachment}}$$

[2] $\lceil x \rceil$ is the ceil function that takes as input a real number and gives as output the least integer greater than or equal to x.

where $\mathsf{N_{attach\_per\_req}} = \mathsf{N_{attachments}}$ for microservices receiving entire emails and $\mathsf{N_{attach\_per\_req}} = 1$ for the others. For *HeaderAnalyser, LinkAnalyser* and *TextAnalyser* we consider $\mathsf{size_{request}}$ to be neglectable, thus (since their pf is also 0) their MCL is infinite. Concerning *MessageAnalyser* request size, we need instead to also consider its MF. In particular, we compute the average size of the MF requests that en email entering the system generates (since we consider only attachments to have a non-negligible size), i.e.

$$\mathsf{size_{request\_MA}} = \frac{\mathsf{N_{attachments}} \cdot (1 - \mathsf{P_V}) \cdot \mathsf{size_{attachment}}}{\mathsf{MF}}.$$

5.3 Calculation of Scaling Configurations

We consider a base **B** system configuration, see Table 1, which guarantees a system MCL of 60 emails/sec. In the corresponding column of Table 1 we present the number of instances for each microservice type, calculated according to the formula in Sect. 2.4. Moreover, we consider four incremental configurations $\Delta 1$, $\Delta 2$, $\Delta 3$ and $\Delta 4$, synthesized via Timed SmartDeployer, each adding a number of instances to each microservice type, see Table 1. Those incremental configurations are used as target configurations for deployment/undeployment orchestration synthesis in order to perform run-time architecture-level reconfiguration. As shown in Table 2, $\Delta 1$, $\Delta 2$, $\Delta 3$ and $\Delta 4$ are used, in turn, to build (summing them up element-wise as arrays) the incremental configurations $\mathsf{Scale1}, \mathsf{Scale2}, \mathsf{Scale3}$ and $\mathsf{Scale4}$ that guarantee an additional system MCL of +60, +150, +240 and +330 emails/sec, respectively.

The reason for not considering our Scales as monolithic blocks and defining them as combinations of the Δ incremental configurations is the following. Let us suppose the system to be, e.g., in a $\mathbf{B} + \mathsf{Scale1}$ configuration and the increase in incoming workload to require the deployment of $\mathsf{Scale2}$ and the undeployment of $\mathsf{Scale1}$. If we had not introduced Δ configurations and we had synthesized orchestrations directly for Scale configurations, we would have needed to perform an undeployment of $\mathsf{Scale1}$ followed by a deployment of $\mathsf{Scale2}$. With Δ configurations, instead, we can simply additionally deploy $\Delta 2$. Moreover, notice that dealing with such an incoming workload increase by naively deploying another $\mathsf{Scale1}$ additional configuration, besides the already deployed one, would not lead the system MCL to be increased of another +60 emails/sec. This is because the maximum number of email per seconds that can be handled by individual microservices composing the obtained $\mathbf{B} + 2 \cdot \mathsf{Scale1}$ configuration would be unbalanced. Such an effect worsens if the system incoming workload keeps slowly increasing and further additional $\mathsf{Scale1}$ configurations are deployed. Since $\mathsf{Scale1}$ for some microservices (*AttachmentManager, ImageAnalyser*) does not provide additional instances, such microservices would eventually become the bottleneck of the system and the system MCL would no longer increase. Moreover, Δ configurations yield, w.r.t. monolithic Scale ones, a finer granularity that makes Timed SmartDeployer orchestration synthesis faster.

For each microservice type, the number of additional instances considered in Tables 1 and 2 for the Scale configurations has been calculated as follows.

Table 1. Base **B** (60 $\frac{emails}{sec}$) and incremental Δ configurations.

Microservice	B	$\Delta 1$	$\Delta 2$	$\Delta 3$	$\Delta 4$	Microservice	B	$\Delta 1$	$\Delta 2$	$\Delta 3$	$\Delta 4$
Message Receiver	1	+1	+0	+1	+1	**Virus Scanner**	1	+1	+2	+1	+2
Message Parser	1	+1	+0	+1	+1	**Attachment Manager**	1	+0	+1	+0	+1
Header Analyser	1	+0	+0	+0	+0	**Image Analyser**	1	+0	+1	+0	+1
Link Analyser	1	+0	+0	+0	+0	**NSFW Detector**	1	+1	+2	+1	+2
Text Analyser	1	+0	+0	+0	+0	**Image Recognizer**	1	+1	+2	+1	+2
Sentiment Analyser	2	+1	+3	+2	+2	**Message Analyser**	1	+1	+2	+1	+2

Table 2. Incremental Scale configurations.

Scale 1 (+60 $\frac{emails}{sec}$)	Scale 2 (+150 $\frac{emails}{sec}$)	Scale 3 (+240 $\frac{emails}{sec}$)	Scale 4 (+330 $\frac{emails}{sec}$)
$\Delta 1$	$\Delta 1 + \Delta 2$	$\Delta 1 + \Delta 2 + \Delta 3$	$\Delta 1 + \Delta 2 + \Delta 3 + \Delta 4$

Given the additional system MCL to be guaranteed, the number $\mathsf{N_{deployed}}$ of instances of that microservice already deployed and its MF and MCL, we have:

$$\mathsf{N_{instances}} = \left\lceil \frac{(\mathsf{base\_sys\_MCL} + \mathsf{additional\_sys\_MCL}) \cdot \mathsf{MF}}{\mathsf{MCL}} - \mathsf{N_{deployed}} \right\rceil$$

In the following section we will present the algorithm for global adaptation. The algorithm is based on the principles described here, i.e., it has the following *invariant* property: if N Scale configurations are considered ($\mathsf{N} = 4$ in our case study) and are indexed in increasing order of additional system MCL they guarantee, the system configuration reached after adapting to the monitored inbound workload is either **B** or $\mathbf{B} + (\mathsf{n} \cdot \mathsf{ScaleN}) + \mathsf{scale}$, or some $\mathsf{scale} \in \{\mathsf{Scale1}, \mathsf{Scale2}, \dots, \mathsf{ScaleN}\}$ and $\mathsf{n} \geq 0$. The invariant property indeed shows, as we explained above, that the deployment of sequences of the same Scale configuration is not allowed, except for sequences of ScaleN. This is because, the biggest configuration ScaleN should be devised, for the system being monitored, in such a way that the inbound workload rarely yields to additional scaling needs. Moreover, even if a sequence of ScaleN occurs, the system would be sufficiently balanced. This is because, differently from smaller Scale configurations, ScaleN is assumed to add, at least, an instance for each microservice having non-infinite MCL (as for Scale4 in our case study).

5.4 Calculation of the Mixed Monitored and Predicted Workload

The fact that predictors are weak against exceptional events is well-known and affects approaches that just rely on predictions: in the case of global scaling, it would result in the execution of inappropriate deployment orchestrations (either wasting resources or degrading the QoS). In this section, we propose a solution mixing proactive and reactive global scaling (reactive and proactive mode of the Actuation Module, see above): we program the Actuation Module to calculate a *target workload* by combining the monitored and predicted ones.

Our algorithm does not rely on comparing the estimated and actual number of inbound requests in a given time window. The reason is that the dynamic interaction between message queues and scaling times makes it difficult to reliably estimate the accuracy of the predicted scaling configuration w.r.t. traffic fluctuations. Thus, we introduce a new, stable estimation, rooted in the workload measure defined below.

Our idea is to use the system MCL of the current configuration (reached by applying some incremental Δ configurations to the base **B** one) and to consider the difference (in terms of number of incremental Δ configurations added) between the system MCL induced by the monitored and predicted traffic. In this way, we can estimate both over- and under-scaling of proactive global scaling.

More precisely, our estimation considers a statically-defined score s for each type Δ configuration, based on the amount of system MCL increment it provides. Following Sect. 5.3, we have $i \in [1,4]$ different Δi applied sequentially (in the exceptional case $\Delta 4$ is not enough, we restart from $\Delta 1$). For each Δi we have a differential system MCL increment of: $\Delta MCL_1 = 60$ for $\Delta 1$ and $\Delta MCL_i = 90$ for Δi with $2 \leq i \leq 4$. Given ΔMCL_i, we compute $s_i = \frac{\Delta MCL_i}{\sum_{j=1}^{4} \Delta MCL_j}$. Notice that this yields $\sum_{i=1}^{4} s_i = 1$.

Then, for each time window tw, we compute our estimation following these 3 steps. In step 1, we calculate, for each index i, the absolute value $|\mathit{diff}_i|$ of the difference between the applications number of Δi induced by the predicted and monitored workload at time window tw. Then, we compute a weight $w \in [0,1]$ that we later use to combine both workloads. Since $|\mathit{diff}_i| > 1$ only happens in exceptional cases (when $\Delta 4$ is not enough), we compute $w = min\left(\sum_{i=1}^{4} s_i \cdot |\mathit{diff}_i|, 1\right)$.

We keep track of the values w computed in the last 3 time windows using function $h = \{(1, w_{tw-2}), (2, w_{tw-1}), (3, w_{tw})\}$, where w_{tw} is the weight computed for the current time window and w_{tw-2}, w_{tw-1} are the preceding ones. The pairs $(1, w_{tw-2}), (2, w_{tw-1})$ are included in h only if the system was already running at those times.

In step 2, we compute the overall weight $w_{ov} = \frac{\sum_{(i,w)\in h} w \cdot i}{\sum_{(i,\_)\in h} i}$ of tw. In particular, $w \cdot i$ means that the most recent w is the most influential one in the sum. The overall weight indicates the distance between the monitored and predicted one. Specifically, the closer the overall weight is to 1 the more distant the prediction is from the monitored workload.

In step 3, we use w_{ov} to linearly combine the predicted and monitored workload to estimate the target workload passed as input to the Adaptation Algorithm $target\_workload = (w_{ov} \cdot monitored\_workload) + ((1 - w_{ov}) \cdot predicted\_workload)$.

5.5 Scaling Algorithm

We now present the algorithm for global adaptation. As a matter of fact, for comparison purposes, we also realized an algorithm for local adaptation simulating the mainstream approach, e.g., Kubernetes [26]. In both of them we use a scaling condition on monitored inbound workload involving two constants called K and k. K is used to leave a margin under the guaranteed MCL, so to make sure that the system can handle the inbound workload. k is used to prevent fluctuations, i.e., sequences of scale up and down.

The condition for scaling up is $(\mathsf{target\_workload} + \mathsf{K}) - \mathsf{total\_MCL} > \mathsf{k}$ and the one for scaling down is $\mathsf{total\_MCL} - (\mathsf{target\_workload} + \mathsf{K}) > \mathsf{k}$. The interpretation of such conditions changes, depending on whether they are used for the local or global adaptation algorithm. In the case of local adaptation the conditions would be applied by monitoring a single microservice type: target_workload would be the number of requests per second received by the microservice load balancer and total_MCL would be the MCL of a microservice instance of that type (calculated as explained in Sect. 2.4) multiplied by the number of deployed instances. In the global adaptation case that we detail in the following, the conditions are, instead, applied by monitoring the whole system: target_workload is the number of requests (emails in our case study) per second entering the system and total_MCL is the system MCL. Notice that the target_workload is computed according to the mode in which the system is used, i.e., reactive (the monitored workload is the target one), proactive (the predicted workload is the target one) and proactive-reactive (mixing the monitored and predicted workload according to the technique presented in Sect. 5.4).

Concerning global adaptation, we have a single monitor that periodically executes the global scaling algorithm presented in code excerpt below. Notice that *kbig*() and *k*() are respectively the K and k constants described above, implemented as constant functions mimicking global variables in the code; scaler is a previously instantianted object that implements the methods *computeConfiguration* and *scale*, presented afterwards.

```
if(target_workload - (mcl - kbig()) > k() || (mcl - kbig()) -
    target_workload > k()) {
  List<Int> target_config = scaler.computeConfiguration(
      target_workload);
  scaler.scale(target_config);
}
```

The *computeConfiguration* method, whose code is presented below, aims at computing the system configuration needed to cope with the *target_workload* passed as input. Such configuration is expressed in the form of a List where index i represents Δi and the *i-th* element is the number of Δi applications.

```
List<Int> computeConfiguration(Rat target_workload) {
  List<Int> configDeltas = this.createEmpty(numScales);
  printableconfig = configDeltas;
  List<Int> config = baseConfig;
  mcl = this.mcl(config);
  Bool configFound = (mcl - kbig()) - target_workload  >= 0;
  while(!configFound) {
    List<Int> candidateConfig = baseConfig;
    Int i = -1;
    while(i < numScales - 1 && !configFound) {
      i = i + 1;
      candidateConfig = this.vSum(config, nth(scaleComponents,i));
      mcl = this.mcl(candidateConfig);
      configFound = (mcl - kbig()) - target_workload  >= 0;
      }
      config = candidateConfig;
      printableconfig = this.incrementValue(i,printableconfig);
      configDeltas = this.addDeltas(i,configDeltas);
    }
    return configDeltas;
  }
```

The code above uses constants numScales, representing the number of Scale configurations (4 in our case study), and scaleComponents: an array[3] of numScales elements (corresponding to Table 2) that stores in each position an array representing a Scale configuration (i.e., specifying, for each microservice, the number of additional instances to be deployed). Moreover, the code uses the variable mcl, containing the current system MCL (assumed to be initially set to the **B** configuration MCL, see Table 1). At first, the code applies the above described scale up/down conditions. Then it loops, starting from the **B** configuration in variable config (an array that stores, for each microservice, the number of instances we currently consider), and selecting Scale configurations to add to config, until a configuration c is found such that its system MCL satisfies $\mathsf{mcl} - \mathsf{K} - \mathsf{target\_workload} \geq 0$. The system MCL of a configuration c is calculated with method *mcl*, which yields

$$\mathsf{min}_{1 \leq \mathsf{i} \leq \mathsf{length(config)}} \ \mathsf{nth(config, i-1)} \cdot \mathsf{MCL_i/MF_i}$$

with $\mathsf{MCL_i/MF_i}$ denoting the MCL/MF of the i-th microservice. More precisely the algorithm uses an external loop updating variables config and configDeltas according to the incremental Scale selected by the internal loop: configDeltas is an array of numScales elements that keeps track of the number of currently deployed Δ incremental configurations (assumed to be initially empty, i.e., with all 0 values). Every time a Scale configuration is selected, configDeltas is updated by incrementing the amount of the corresponding Δ configurations (as described in Table 2). The internal loop selects a Scale configuration by looking for the first one that, added to config, yields a candidate configuration whose system MCL satisfies the condition above. If such Scale configuration is not found then it just selects the last (the biggest) Scale configuration (Scale4 in our case study), thus implementing the invariant presented in Sect. 5.3.

The *scale* method presented below enacts the scaling operations required to reach the system configuration passed as input.

```
Unit scale(List<Int> configDeltas) {
  Int i = 0;
  while(i < numScales) {
    Int diff = nth(configDeltas,i) - nth(deployedDeltas,i);
    Rat num = abs(diff);
    while(num > 0) {
       if (diff > 0) {nth(orchestrationDeltas,i)!deploy();}
       else {nth(orchestrationDeltas,i)!undeploy();}
       num = num - 1;
    }
```

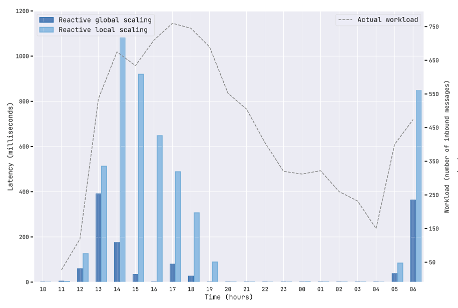
[3] The ABS instructions nth(a, i) and length(a) retrieve the i-th element and the length of the a array, respectively.

```
        i = i + 1;
      }
      deployedDeltas = configDeltas;
      scalingAct = this.recordAction(scalingTrace, printableconfig);
      scalingTrace = printableconfig;
    }
```

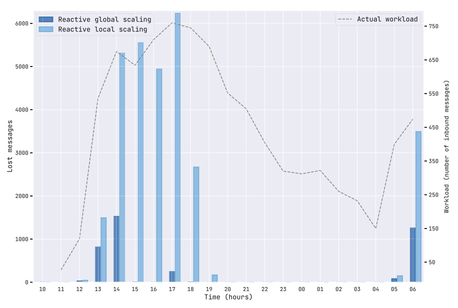
Given the target Δ configurations configDeltas to be reached and the current deployedDeltas (an array with the same structure of configDeltas) ones, the *scale* method performs the difference between them so to find the Δ orchestrations that have to be (un)deployed. We use methods *deploy/undeploy* of the object in the position $i-1$ of the array orchestrationDeltas to execute the orchestration of the i-th Δ configuration. In our model such an orchestration is the ABS code generated by Timed SmartDeployer at compile-time: it makes use of ABS primitives *duration(...)* and *decrementResources(...)* to *dynamically* set, respectively, the overall startup time to the maximum of those of deployed DCs and the speed of such DCs accounting for the virtual cores actually being used (by decrementing the DC static speed, see Sect. 3.2). In this way we are guaranteed that each microservice always preserves the desired fixed MCL we want to model (see Sect. 2.4). Moreover, we remind that, besides speed, also constraints related to other resources (memory) are considered in the Timed SmartDeployer synthesis process. Notice that the variables scalingAct, scalingTrace as well as the *recordAction* method are only used for debug purpose.

5.6 Benchmarking the Performance of Global Scaling Approaches

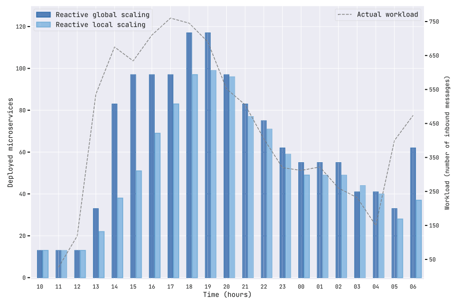
In this section we present simulation results obtained with our ABS programs [1] modeling reactive local scaling and the three variants of the global one, i.e., reactive, proactive and proactive-reactive. In particular, at first, we show the impact of reactive global scaling on system performance w.r.t. the reactive local one; then we show how the reactive global scaling can be further improved endowing it with proactive capabailities, e.g., making use of a workload predictor. Finally, we show the risks of just relying on workload predictions to enact scaling actions and the need of mixing reactiveness and proactiveness. We make use of (part of) the Enron dataset [28] as the inbound workload inputed to the simulations, to test the performance of reactive and proactive global scaling and the local one. All benchmark tests shown in this section are performed on email traffic on a weekday in May 2001. To prove the effectivenss of our proactive-reactive global scaling, we selectively picked outliers from the Enron dataset to produce a traffic flow that our predictor would struggle to forecast, thus the workload inputed to this simulation differs from the one inputed to the others. In our simulations we consider the following metrics: (i) latency (considered as the average time for completely processing an email that enters the system), (ii) message loss, (iii) number of deployed microervices and (iv) monetary costs. Notice that in the comparison between reactive local scaling and the reactive global one, we do not consider monetary costs, since Timed SmartDeployer orchestrations are such that costs are minimized.



(a) Latency



(b) Lost messages

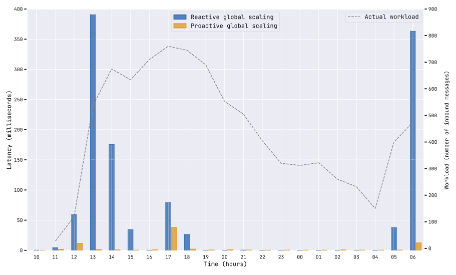


(c) Deployed microservices

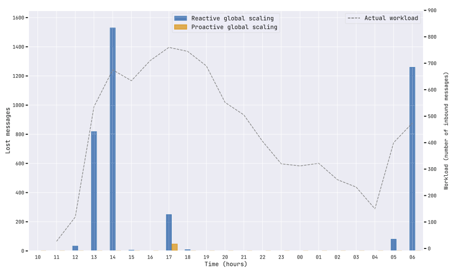
Fig. 4. Comparison between reactive local scaling and the reactive global one.

Reactive Local vs Reactive Global Scaling. Considering the flow of incoming emails in the workload inputed to the simulation, it is clear the extent of the improvement introduced by our approach: our global adaptation [10] makes the system adapting much faster than the local approach. This is caused by the ability of the global adaptation strategy of detecting in advance the scaling needs of all system microservices. This is shown in Figs. 4a and 4b, where our reactive global scaling approach outperforms the local one: latency and message loss are restored in much shorter time w.r.t. the reactive local scaling.

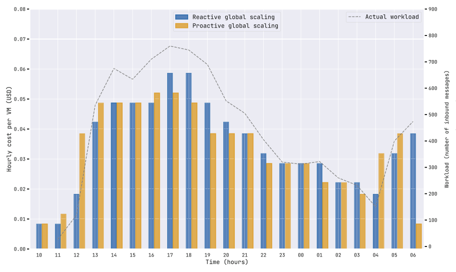
Comparing the number of deployed microservices helps to have a deeper understanding of the reasons why the global adaptation performs better. As shown in Fig. 4c, our approach reaches the target configuation, needed to handle the monitored workload, faster than the local scaling approach. As expected, this makes the adaptation process slower and worsens the performance. The local adaptation slowness in reaching such a target configuration is caused by a scaling chain effect: local monitors periodically check the workload, thus single services scale one after the other. Hence, w.r.t. global adaptation, where the architecture is replicated as a single block, the number of instances grows slower. For example, considering the attachment pipeline in Fig. 2, the first microservice to become a bottleneck is the *Virus Scanner*: it starts losing requests, which will never arrive to the *Attachment Manager*. Therefore, this component will not



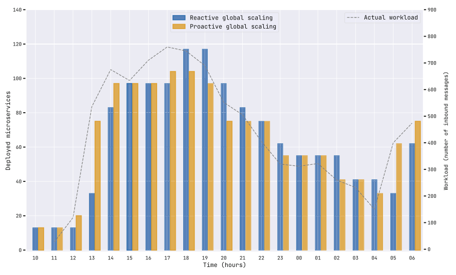
(a) Latency



(b) Lost messages



(c) Virtual machine cost-per-hour



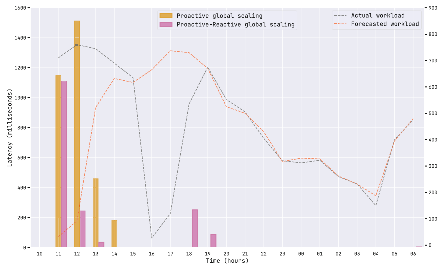
(d) Deployed microservices

Fig. 5. Comparison between reactive and proactive global scaling approaches.

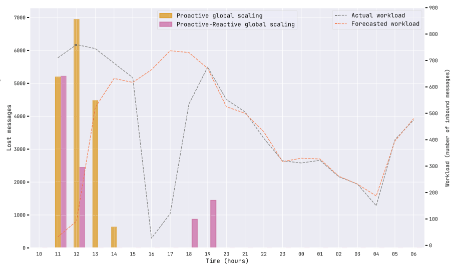
perceive the increment in the inbound requests until the *Virus Scanner* will be replicated, thus causing a scaling chain effect that delays adaptation. This is the main cause for the large deterioration in performances observed.

Proactive vs Reactive Global Scaling. To give an intuition of the effectiveness of our proactive global scaling approach [9], we test its performance against reactive global scaling [10]. This comparison mainly aims at showing the improvement brought in the global scaling technique thanks to the use of a workload predictor, i.e., endowing it with proactive capabilities.

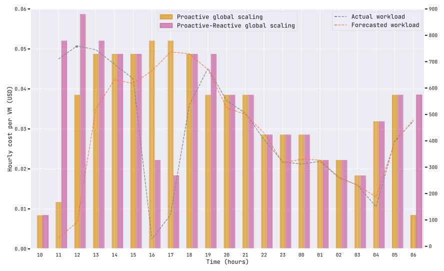
Considering latency, as shown in Figs. 5a and 5b, the proactive scaling is barely visible given that it performs in advance the scaling operations needed to manage workload peaks, with negligible latency. The small visible spikes are imputable to inaccuracies in the workload predictions. On the other hand, the reactive approach suffers the most at sudden peaks of workload because of the time needed to complete the adaptation process, e.g., VMs startup time. As seen in Figs. 5c and 5d, despite the signifante difference in performance, the costs/number of deployed instances are the same, although shifted by a time-unit backwards. The reason is that, since the traffic is the same, resource (de)allocations are the same across all the approaches, although these happen one time-unit in advance in the proactive case.



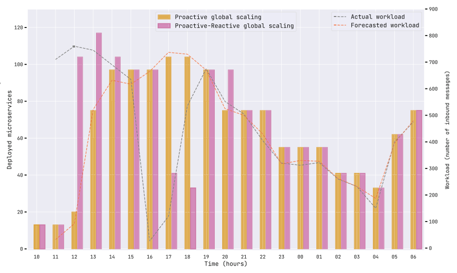
(a) Latency



(b) Lost messages



(c) Virtual machine cost-per-hour



(d) Deployed microservices

Fig. 6. Comparison between proactive-reactive and proactive global scaling, on the outliers test set.

Proactive-Reactive vs Proactive Global Scaling. The presented proactive approach proved to be quite effective. However, predictors are not infallible: if the traffic greatly deviates from the historical data, due to some unprecedented occurrence, the predictor can fail to provide an accurate estimation of the traffic. This fact, considered in the context of proactive global scaling (like the one implemented above) where scaling decisions neglect actual traffic fluctuations, can result in over- (wasted resources) or under-scaling (latency, request loss) of the system. To illustrate how much this phenomenon can affect performance, we selectively picked outliers from the test set described in Sect. 4.2 and used these to produce a traffic flow that our predictor struggles to forecast. From Figs. 6a and 6b, the proactive-reactive global scaling rapidly recovers from wrong predictions, while the proactive one neglects unexpected traffic fluctuations. This is visible, e.g., in the interval 11–13, where the proactive approach expects fewer requests and endures high latency. Also the proactive-reactive global scaling initially undergoes high latency, but, detecting the diverge with the predictions, it assumes a reactive stance and quickly adapts. Note that the latency of the proactive-reactive approach in the timespan 18–19 is “good”. Indeed, while the workload drops between 15–17, the proactive approach allocates a high number of microservices (cf. Figures 6c and 6d), wasting resources. Contrarily, the other one (reacting to unforeseen changes) trades some minor latency off resource savings.

6 Related Work and Conclusion

We have presented an integrated approach for the design, specification, automatic deployment, and simulation of microservice architectures, based on the ABS language. The basic ingredients of this approach are:

– the ABS language, used to specify the behaviour of microservices;
– deployment annotations added to the ABS code, carrying information like the available computing resources and their costs, the resources consumed by each microservice instance, and constraints about the minimum number of instances for each microservice;
– the use of a compile-time deployment engine able to synthesize optimal deployments starting from deployment annotations extracted from ABS code;
– compilation of timed ABS code into executable Erlang program, to simulate the specified system.

To the best of our knowledge, our approach is the only one mixing *a*) formal specification of microservice behaviour, *b*) the usage of a language equipped with executable semantics for simulation and performance analysis, and *c*) the modeling and automatic synthesis of deployment orchestrations. Specifically, related work addressed the above aspects separately. Concerning executable semantics for simulation, [12] instead of compiling ABS into Erlang, makes use of a real-time Haskell backend: this makes it possible for the simulation to communicate with real services, thus mixing external execution and simulation at run-time. In our case, the communication between the simulated system and external systems (during simulation) is not needed, thus we avoid the complexities of the approach in [12] related to synchronizing real and simulated time. Another line of work encompasses the usage of timed/stochastic process algebras by integrating them in the software development process, with the aim of analysing the performances of the modeled system (see, e.g., the surveys [11,25]). Finally, other proposals adopt specific models for cloud deployment specification, e.g., TOSCA (Topology and Orchestration Specification for Cloud Applications) [30] or AEOLUS [21], to describe the components of a cloud service system and their deployment/orchestration process. The interested reader can find a recent survey of the model-based methodologies used to ensure the correctness of reconfigurations in component-based systems at [19].

In this presentation, we applied our integrated approach to the analysis of different techniques to deal with the problem of dynamic scaling of microservices applications. In particular, we have considered a rather sophisticated technique based on a mixture of predicted and monitored inbound workload, with subsequent global adaptations of the entire system (i.e., all the microservices that will be influenced by the modified workload will coordinate their scaling). A similar technique has been already investigated by Urgaonkar et al. [32]. Differently from our approach, [32] only focuses on adjusting under-estimations of the actual workload, to guarantee a given QoS. In the case of over-estimation, their approach simply considers the predicted workload as the target one, ending up wasting resources (and money), see [9] for a detailed comparison.

References

1. Code repository for the email processing examples. https://github.com/LBacchiani/ABS-Simulations-Comparison
2. Ábrahám, E., Corzilius, F., Johnsen, E.B., Kremer, G., Mauro, J.: Zephyrus2: on the fly deployment optimization using SMT and CP technologies. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 229–245. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_15
3. ABS. ABS documentation. http://docs.abs-models.org/
4. ABS. ABS toolchain. https://www.sciencedirect.com/science/article/pii/S0167642322000946
5. ABS. Core ABS. https://www.sciencedirect.com/science/article/pii/S2352220814000479
6. ABS. Deployment component in ABS. https://link.springer.com/chapter/10.1007/978-3-642-25271-6_8
7. ABS. Real time ABS. https://link.springer.com/article/10.1007/s11334-012-0184-5
8. Amazon, AWS auto scaling. https://aws.amazon.com/autoscaling/
9. Bacchiani, L., Bravetti, M., Gabbrielli, M., Giallorenzo, S., Zavattaro, G., Zingaro, S.P.: Proactive-reactive global scaling, with analytics. In: Troya, J., Medjahed, B., Piattini, M., Yao, L., Fernández, P., Ruiz-Cortés, A. (eds.) Service-Oriented Computing - 20th International Conference, ICSOC 2022, Seville, Spain, 29 November–2 December 2022, Proceedings, vol. 13740 of Lecture Notes in Computer Science, pp. 237–254. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-20984-0_16
10. Bacchiani, L., Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Microservice dynamic architecture-level deployment orchestration. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 257–275. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_16
11. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: a survey. IEEE Trans. Softw. Eng. **30**(5), 295–310 (2004)
12. Bezirgiannis, N., de Boer, F., de Gouw, S.: Human-in-the-loop simulation of cloud services. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 143–158. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_11
13. Binder, W., Hulaas, J., Camesi, A.: Continuous bytecode instruction counting for cpu consumption estimation. In: Third International Conference on the Quantitative Evaluation of Systems-(QEST 2006), pp. 19–30. IEEE (2006)
14. Binder, W., Hulaas, J., Moret, P., Villazón, A.: Platform-independent profiling in a virtual execution environment. Softw. Pract. Exp. **39**(1), 47–79 (2009)
15. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of asynchronous session subtyping. Inf. Comput. **256**, 300–320 (2017)
16. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Optimal and automated deployment for microservices. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 351–368. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_21
17. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: A formal approach to microservice architecture deployment. In: Microservices, pp. 183–208. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-31646-4_8

18. Di Cosmo, R., Zacchiroli, S., Zavattaro, G.: Towards a formal component model for the cloud. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 156–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_11
19. Coullon, H., Henrio, L., Loulergue, F., Robillard, S.: Component-based distributed software reconfiguration: a verification-oriented survey. ACM Comput. Surv. **56**(1), 1–37 (2023)
20. de Gouw, S., Mauro, J., Zavattaro, G.: On the modeling of optimal and automatized cloud application deployment. J. Logical Algebr. Methods Program. **107**, 108–135 (2019)
21. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Inf. Comput. **239**, 100–121 (2014)
22. Docker. Docker compose documentation. https://docs.docker.com/compose/
23. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
24. Fromm, K.: Thinking Serverless! How New Approaches Address Modern Data Processing Needs. https://medium.com/a-cloud-guru/thinking-serverless-how-new-approaches-address-modern-data-processing-needs-part-1-af6a158a3af1
25. Hermanns, H., Herzog, U., Katoen, J.-P.: Process algebra for performance evaluation. Theor. Comput. Sci. **274**(1–2), 43–87 (2002)
26. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running Dive into the Future of Infrastructure, 1st edn. O'Reilly Media Inc., Sebastopol (2017)
27. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional, Boston (2010)
28. Klimt, B., Yang, Y.: The enron corpus: a new dataset for email classification research. In: Machine Learning: ECML 2004, Berlin, pp. 217–226 (2004)
29. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
30. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html. Accessed May 2020
31. Rawdat, A.: Testing the performance of nginx and nginx plus web servers. https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/
32. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., Wood, T.: Agile dynamic provisioning of multi-tier internet applications. ACM Trans. Auton. Adapt. Syst. (TAAS) **3**(1), 1–39 (2008)