The Journal of Systems & Software 169 (2020) 110701

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# Architecture design evaluation of PaaS cloud applications using generated prototypes: PaaSArch Cloud Prototyper tool

David Gesvindr [a,*], Ondrej Gasior [a], Barbora Buhnova [a,b]

[a] Faculty of Informatics, Masaryk University, Brno, Czech Republic
[b] Institute of Computer Science, Masaryk University, Brno, Czech Republic

ARTICLE INFO

*Article history:*
Received 7 August 2019
Received in revised form 10 May 2020
Accepted 16 June 2020
Available online 23 June 2020

*Keywords:*
Cloud computing
Software architecture design
Prototype generation
Quality evaluation
Performance
Internet of things

ABSTRACT

Platform as a Service (PaaS) cloud domain brings great benefits of an elastic platform with many prefabricated services, but at the same time challenges software architects who need to navigate a rich set of services, variability of PaaS cloud environment and quality conflicts in existing design tactics, which makes it almost impossible to foresee the impact of architectural design decisions on the overall application quality without time-consuming implementation of application prototypes. To ease the architecture design of PaaS cloud applications, this paper proposes a design-time quality evaluation approach for PaaS cloud applications based on automatically generated prototypes, which are deployed to the cloud and repeatedly evaluated in the context of multiple quality attributes and environment configurations. In this paper, all steps of the approach are described and demonstrated on an example of a real-world complex IoT system for collection and processing of Smart Home sensor data. The approach has been implemented and the automated prototype generation and evaluation tool, referred to as PaaSArch Cloud Prototyper, is presented together with the approach.



## 1. Introduction

Over the past decade, cloud computing has become a dominant model to operate applications, which has stimulated global transformation of IT industry. Cloud computing is also changing the way we operate our current applications, and motivates the design of cloud-native applications, allowing developers to eliminate investment costs, and take advantage of cloud elasticity, while getting billed only for the demanded amount of resources. There are currently three service models in the Cloud: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). At the moment, many software developers build on their knowledge from on-premise environment and end up employing the IaaS model, which however only provides them with basic computational resources (virtual servers, storage and software defined network).

By opting for the PaaS cloud to operate the designed application, software developers can take advantage of research and development done by the cloud provider and use platform services that are built on top of IaaS, to develop their applications faster (up to 50% time to market reduction (Anon, 2016)) with lower costs and operate them without servicing the platform as they would need to do in case of IaaS.

From the design perspective, the employment of the PaaS cloud changes the way of thinking about software architecture design. So far, software architects could take advantage of architectural and design patterns that were validated in on-premise environment, and are with minor modifications transferable to the IaaS environment. Similarly, they could employ design-time prediction tools, meant to estimate various quality attributes of the designed application. All these however suffer from crucial deficiencies when applied to PaaS cloud applications, because of the following challenges associated with the PaaS cloud:

- **Hidden complexity of the platform** — The inner PaaS cloud service architecture is highly complex, with many computation nodes, shared load balancers, partitioning of data, and other optimizations, to assure its high availability, scalability and elasticity in combination with the complex functionality of provided services. Due to the lack of publicly available information about the inner architecture of the PaaS services, together with the frequent updates, and hidden resource limits of the infrastructure unavailable to the architect, it is practically impossible to model such a service ecosystem and make it part of a PaaS cloud application model that could serve as the basis for model-based quality assessment.
- **Multi-tenant environment** — It is common in PaaS cloud to have the application hosted in a shared environment,

* Correspondence to: Faculty of Informatics, Masaryk University, Botanická 68a, Brno, Czech Republic
*E-mail addresses:* gesvindr@mail.muni.cz (D. Gesvindr), 448273@mail.muni.cz (O. Gasior), buhnova@mail.muni.cz (B. Buhnova).

https://doi.org/10.1016/j.jss.2020.110701
0164-1212/

where although being strictly isolated from other applications running on the same server (due to security reasons), the application is sharing the computational resources of the server. Then although the application is provided with dedicated virtual servers managed by the cloud operator, it needs to rely on shared network infrastructure, load balancers, and other services, which may affect the validity of the model-based assessment results due to the presence of noisy neighbors.

- **Updates of the environment** — Cloud environments are undergoing constant changes, including hardware upgrades and software optimization of their services, often on daily or weekly basis. Therefore, any model of the PaaS cloud architecture mapping the PaaS cloud platform specifics quickly becomes outdated. Both Microsoft Azure and Amazon Web Services currently offer over 100 cloud services (Anon, 2017c,b). Construction and frequent updates of a model mapping these platforms would be highly ineffective.

Given these aspects that make it practically impossible to estimate PaaS cloud application performance and other characteristics without its actual deployment to the cloud, software architects are nowadays relying on costly manual implementation of application prototypes deployed to the PaaS cloud. When the development is completed, cloud resources need to be allocated, sample data generated (which may require implementation of sample data generator), and finally the prototype needs to be deployed to the cloud, benchmarked and tested. After the tests are conducted, software architect can make informed decision about a set of services used in the architecture of the application, or re-run the process if more information is needed or other alternatives need to be assessed. When the prototype development is done manually, it is highly tedious and time-consuming. An automated support is however unavailable.

In this paper, we build upon our experience with PaaS cloud application design (Gesvindr and Buhnova, 2016b,a), and introduce an approach and tool set, called PaaSArch Cloud Prototyper[1], supporting automated prototyping of PaaS cloud applications, enabling quick assessment of various architectural options integrating different PaaS cloud services. The tool receives a model of the application architecture (together with some details about the inner behavior of the application and its usage of cloud services), and translates it via an automated process into a source code of a fully functional application prototype, which is together with automatically generated sample data deployed to the cloud and benchmarked.

The approach is designed to enable quality evaluation of most common PaaS cloud applications, which are web applications hosted using PaaS cloud services with a logic executed on the backend server frequently communicating with other cloud services (storage, queue, etc.). The current implementation of the approach support generation of two most common types of applications within this context, which are REST API applications and worker applications. Regarding the cloud provider, the tool at the moment implements the support of Microsoft Azure cloud, one of the market leaders in cloud computing, whose services are in parity with other providers such as Amazon. Although so far the tool only supports Azure, our prototyping approach is independent of a specific cloud provider as all of the providers share the same fundamental principles and challenges of the PaaS cloud described earlier. Furthermore, we have designed the architecture of our prototyping tool with rich extensibility, to be ready to support other cloud providers at the equivalent level to the current support of Microsoft Azure.

This paper is an extended version of Gesvindr et al. (2017a), which introduced the first version of the approach, and Gesvindr and Buhnova (2019), which presented very short introduction of the tool itself. In this paper, we present an elaborated version of the approach and the tool, which on top of Gesvindr et al. (2017a) and Gesvindr and Buhnova (2019) includes full automation of cloud resource allocation, elaborated data generation and query optimization reflecting the generated data, full extensibility support and much greater maturity of the whole approach, including a detailed case study. Over the past two years, the approach has been validated on multiple real-world case studies, from which this paper details a case study of an IoT home automation system, which very well demonstrates that if the software architect is well guided, the solution architecture might be cheaper and more efficient than an intuitive solution using predesigned IoT architecture by the PaaS cloud provider.

The paper is structured as follows. After the discussion of related work in Section 2, Section 3 outlines our approach, which is detailed in sections dedicated to the modeling (Section 4), prototype generation (Section 5), deployment and evaluation (Section 6). The whole approach is illustrated on a case study in Section 7, and the implementation of the PaaSArch Cloud Prototyper tool is discussed in Section 8. After discussion in Section 9, we conclude in Section 10.

## 2. Related work

One of the most popular ways of preventing poor early design decisions, including architecture-relevant design decisions, is based on design patterns (Gorton, 2006). A rich set of design patterns has been designed for instance by Gamma et al. (1995) or Fowler (2002), but as these are not meant for PaaS cloud, they are not taking the rich PaaS service portfolio into account. Nor they consider trade-offs of PaaS cloud services, which is why they need to be validated prior their use. There are new catalogs of design patterns for the cloud emerging (Erl et al., 2013; Wilder, 2012; Homer et al., 2014; Pahl et al., 2018), but impact of their combination on various quality attributes (Bass et al., 2003; Taylor et al., 2009) is not well described.

Since patterns only navigate towards presumably good solution without giving more insight into the effect of patterns usage, Model-Driven Quality Predictions (MDQP) have gained importance as a way to predict and quantify the quality of the designed application. In this family of approaches, the application is first modeled using UML or domain specific languages, e.g. Palladio Component Model (PCM) (Becker et al., 2009). Then the model of the architecture is automatically transformed into a predictive formal model (e.g. Markov Chain or Layered Queuing Network for performance prediction) surveyed for instance in Aleti et al. (2013), Koziolek (2010). There exist some extensions of traditional approaches that are directed towards cloud, like SPACE4CLOUD (Franceschelli et al., 2013), SimuLizar (Lehrig et al., 2015) and CloudSim (Barbierato et al., 2019). Another example is CloudScale (Brataas et al., 2013), which is translating models of cloud applications to PCM models working with dedicated virtual machines (IaaS) and making performance predictions based on sets of conducted measurements of the environment. There are also tools providing support for IaaS cloud simulation (Buyya et al., 2009), while the PaaS cloud services are however not supported.

Another approach to analyze the quality of the application is to deploy the application and run a set of benchmarks. The benefit of real execution is much better approximation of real resource usage in the final execution environment (Chen et al.,

[1] Project homepage: https://lasaris.fi.muni.cz/research/soft\discretionary{-}{}{}ware-architecture-optimization-in-paas-cloud-applications/paasarch-cloud-prototyper-tool

2015). However, the approaches need to handle missing implementation and platform details. The StressCloud project (Chen et al., 2015) overcomes the problem by generation of synthetic workload based on modeled CPU, memory and IO utilization using deployed agents in virtual machines (IaaS), but this tool is not designed for PaaS cloud as its agents require fully featured virtual machines (IaaS).

Instead of synthetic workload, we can generate prototypes of applications from their model. This approach is used by ProtoCom (Klaussner and Lehrig, 2014), which generates prototypes of Java applications. Initial attempts to support variety of platforms (Becker, 2008) remained only at conceptual level and does not yet provide transformation implementations as stated by the authors in Klaussner and Lehrig (2014). Generated prototypes can be hosted in the PaaS cloud (as web applications), but they are not leveraging the full potential of all available platform services.

Deployment of applications to multiple clouds can be modeled using CloudML (Ferry et al., 2018), where the deployment plans can be generated and executed using the models@runtime environment (Ferry et al., 2018). This framework is however intended for deployment of already developed components, and thus is not suitable during design time for generation and evaluation application prototypes.

Tools for web application scaffolding (Anon, 2017a) might be used for prototype generation, but are only helpful when there is high stress on well designed user interface with basic CRUD operations and fixed software architecture over a given data source (mostly relational databases). Their benefit is much smaller in scenarios integrating multiple PaaS services.

In practice, there are two major observable trends beyond PaaS cloud. The first is the rapid expansion in the cloud event technologies followed with an increasing trend in the use of serverless cloud services, which are an extension of the PaaS cloud providing per second consumption billing and built-in automatic scalability. The second is that the software architecture of cloud applications is in a growing scale taking advantage of the decomposition of applications into microservices supported with the increasing use of containers. Progress in the research field of software architecture and cloud computing follows both trends. There is an increasing interest in cloud event technologies (McGrath et al., 2016), serverless (van Eyk et al., 2018) and microservices (Pérez et al., 2018; Bogner et al., 2019). However, despite the expansion of the PaaS cloud into new fields and forms, there is still lack of design guidance for PaaS cloud, serverless, and microservice applications with clearly evaluated performance impacts. The emerging modeling and simulation approaches in the research of architecture design support are not targeted to the PaaS cloud, likely due to the presented challenges, thus leaving still significant space for further research in this domain.

## 3. Our approach

Our approach introduced in this section allows software architects to evaluate the quality of proposed software architecture of a PaaS cloud application in an early design stage of the development process, to prevent later costly reimplementation when the design does not meet desired quality requirements. We take advantage of automatically generated prototypes of PaaS cloud applications based on their model, which are deployed to the PaaS cloud where their quality can be measured and benchmarked.

Our primary focus is the performance and its specific quality attributes, namely the throughput, response time and scalability, as they are frequently specified in application requirements, and influenced by software architecture of the application. Moreover, design-time prediction methods for these attributes in the context of PaaS cloud are so far missing.

In this work, we refer to the definition of *throughput* as a measure of the amount of work that an application must perform in a unit of time (Gorton, 2006). Throughput is being quantified with a number of operations, transactions or requests that the system handles per second or other time unit.

*Response time* is then defined as a measure of the latency an application exhibits in processing a business transaction (Gorton, 2006). For a cloud application, it is important to distinguish between network latency and request processing time. The former is a delay in the communication between the client and the data center where the service is hosted, the latter is influenced by the application architecture and response times of all cloud services that are invoked during request processing.

The core performance characteristics, namely the *throughput* and *response time*, specify the application behavior for a static instance of cloud environment configuration. On the other hand, *scalability* of an application defines how the application is expected to behave with increasing load and dynamic server resources, and whether it is able to effectively utilize newly provided resources.

The main reason why we generate a prototype application, deploy it to the cloud and evaluate it during its execution, instead of using predictive simulation, is because it allows us to assess the application within the current state and version of the cloud environment, and thus to obtain more realistic results. Quality of PaaS cloud services used by the prototype significantly influence overall measured quality attributes of the evaluated prototyped application. It would be highly inefficient to build for this purpose a simulation model, as it would need to reflect the PaaS cloud services, and thus require extensive, time-consuming and costly evaluation of the performance of many cloud services with unknown architecture. Moreover, the validity of such a model would be very limited as the services are without further notice being frequently updated by the cloud provider.

Since the effectiveness of the approach highly depends on an automation of the entire process to enable quick assessment of an architectural alternative, we have implemented a fully-automated prototype generation tool described in this paper, called PaaSArch Cloud Prototyper, which became the backbone of our approach. The tool receives a model of the application architecture (together with some details about the inner behavior of the system and its usage of cloud services), which is via a fully automated process translated into a source code of a fully functional application prototype, which is together with automatically generated sample data deployed to the cloud and ready for bechmarking. The approach constitutes of the following steps that are depicted in Fig. 1 and detailed in this and the next sections.

1. ***Model creation:*** First, the application model needs to be created by the software architect, which matches the software architecture design of the application undergoing the evaluation and is created in accordance to our meta model. The application model describes all important aspects of the application that have impact on the evaluated quality metrics. Structure and flexibility of the model are described in Section 4.
2. ***Model analysis and validation:*** The created application model is loaded in our tool, which analyzes and validates the model. The input model is validated against our meta model to identify unsupported use of elements in the model. It is also checked that our mapping function can map every element in the model, which leads to code-generation in our tool, otherwise the model could not be transformed into a prototype.

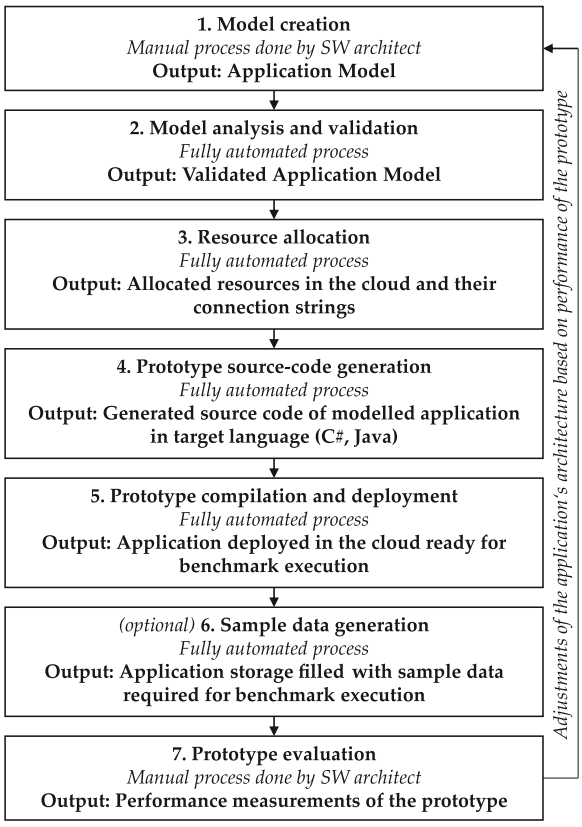


**Fig. 1.** Prototyping process.

3. ***Resource allocation:*** One of the key advantages of our approach is that it deals with unpredictable behavior of the cloud environment by allocation and use of real cloud resources, not just their mathematical approximation. The resource allocation step supports software architect in automating the resource management based on the information provided by the model, which leads to allocation and configuration of all resources and services utilized by the prototypes. Connection strings of allocated resources are passed to the source code generation step.
4. ***Prototype source-code generation:*** The source code of the modeled application is automatically generated based on the model. The process ensures maximum precision of the model-to-prototype generation to secure realistic results during the evaluation phase. The process of code generation is described in Section 5.
5. ***Prototype compilation and deployment:*** Generated source code of the prototyped application is automatically compiled into a form of an executable application, which is automatically deployed to the cloud utilizing allocated resources as specified in the model.
6. ***Sample data generation:*** In order to achieve precise approximation of a real application, the prototype must be able to utilize storage services with the same level of data load as the final application. Thus these data sets are automatically generated in our approach based on the model, so that the storage meets the requirements specified in the application model.
7. ***Prototype evaluation:*** The usual way to communicate with a cloud-deployed application is using HTTP via exposed endpoints, which can later be evaluated and benchmarked. In addition to the application prototype, the PaaSArch Cloud Prototyper generates a list of URLs corresponding to individual actions in the model that can be used during the evaluation process to measure the performance of the prototype. The benchmarks are manually executed by the user of the tool using state-of-the-art standalone HTTP benchmarking tools, independent of our tool with additional details provided in Section 6.4. Based on the benchmark results, the software architect can decide if the architecture meets the expectations and the impact of applied architectural tactics and patterns is desirable, or another design alternative shall be evaluated. Results obtained by evaluation of the prototype are dependent on both the specific cloud provider and the software architecture of the prototype. Despite the fact that cloud providers provide services with a similar functionality (web hosting, relational database, etc.), they provide different levels of performance and scalability due to differences in their internal architecture.

Our approach is applied in terms of an iterative process. The software architect first proposes a single or multiple variants of the designed architecture. These variants are created primarily based on expert knowledge of the architect, but can as well be generated automatically using existing methods, such as Parra et al. (2010). Then the architect models these variants according to the given meta model and with the help of the PaaSArch Cloud Prototyper, obtains information about their quality (performance characteristics in our case) and uses this information for additional tuning of the architecture. With every change in the application's architecture, the model is updated, prototype regenerated, redeployed, reevaluated, and compared with the expectations that shall be met.

Individual steps of the approach and their implementation are described in detail in the following sections. Section 4 describes the creation and validation of the model, Section 5 explains the mechanism of prototype generation, Section 6 details the process of resource allocation, sample data generation and evaluation of the prototype. All these sections are accompanied with practical demonstration of the approach on a real-world case study introduced in the next section.

## 4. Application model

The design of the meta model which is an abstraction of the prototyped application, is one of the essential parts of the approach. In this section we detail the requirements we have on this meta model, introduce the components of the meta model and explain model representation recognized by our PaaSArch Cloud Prototyper tool.

### 4.1. Model requirements

We have identified the following requirements that the meta model has to fulfill in order to support the generation of a prototype with relevant quality characteristics.

- **Use in an early design stage** – The model must build upon the information that is (or might be) available to the software architect during early design stages of the application, so that the software architect can model the application and evaluate the impact of various design decisions before advancing with the implementation. It implies that it is necessary to omit and simplify details of application's design without depreciation of quality evaluation results.

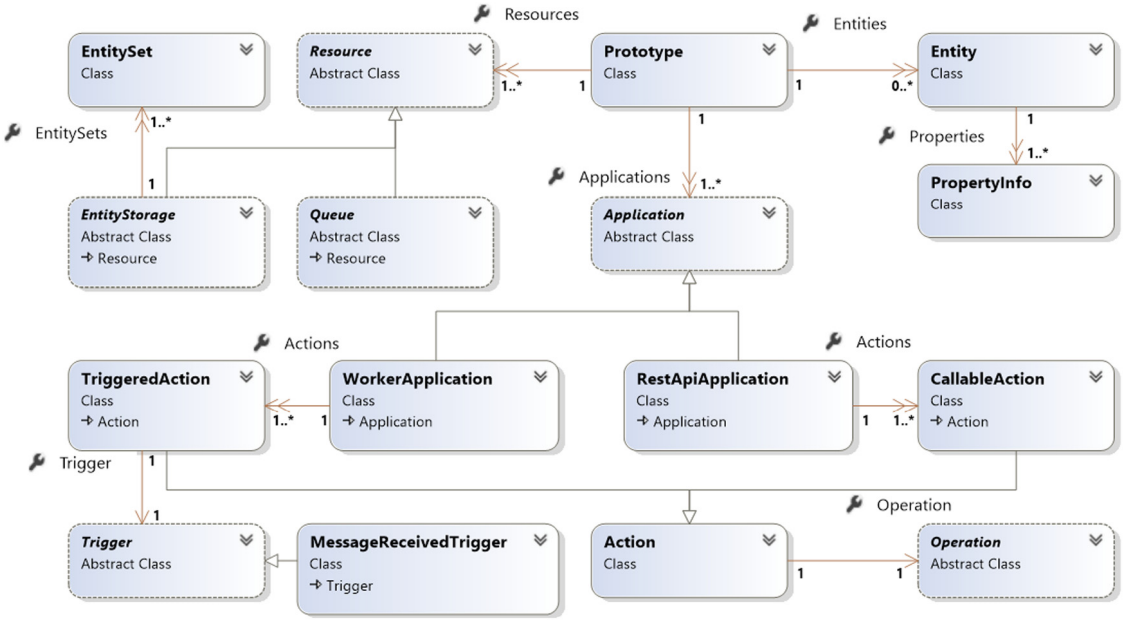


**Fig. 2.** Prototype metamodel.

- **Abstracting from the cloud provider** — Making the application model dependent on PaaS cloud services of a single cloud provider would significantly limit its usability and the number of potential users who can benefit from our work. It is important for us to design platform- and cloud-independent meta model. On the other hand, while maintaining platform independence, it is also necessary to include platform specific details so that the used services can be utilized up to their full potential and efficiency.
- **Multiple types of applications** – Based on our experience with application design, we have identified two major types of modern cloud applications – REST APIs and worker processes. The first essentially processes a request and generates an output that is sent to the client (serialized data entity, HTML page, etc.), the latter is running in an infinite loop or is activated by specific trigger (e.g. arrival of a message to a queue). In both cases the execution of actions leads to consumption of cloud resources.
- **Identical resource consumption** — Performance of PaaS applications is significantly influenced by consumption of other PaaS cloud services (e.g. storage services such as relational databases, NoSQL databases), where consumption of a PaaS cloud service can be described as a remote call to a PaaS service of a given type, frequency and executed operation. As we described in our previous work Gesvindr and Buhnova (2016b), all three factors have significant impact on the throughput, response time and scalability of the designed application, and needs to be part of our meta model.
- **Similar data complexity** — We have identified that the amount of data loaded from data sources, processed and sent to the client, has measurable effect on performance of the used PaaS cloud services. Therefore it is necessary to use data entities in the model that reflect the complexity of the data in the designed application. In an early design stage of the application, the details about employed data entities and data transfer objects are usually not available, therefore the meta model should work with a degree of complexity of the given data entity, rather than with precise description of all attributes.
- **Model extensibility** — The meta model needs to be easily extensible, so that both new abstract and platform-specific resources and operations can be easily added based on specific use-cases of modeled applications, which were not considered in the meta model we currently provide.
- **Support for modeling of multiple interacting applications** — It is necessary to support modeling of multiple interacting applications to make this meta model applicable for analysis of impacts of deployment across multiple data centers or studies of micro service architectures.
- **Abstracting from the programming language** – Applications should be modeled independently of the programming languages and related frameworks that will be used for their development. This gives the software architect an opportunity to easily evaluate performance differences in frameworks that are considered to be used.

Based on the defined requirements, we have designed a meta model described in the rest of this section.

### 4.2. Components of the meta model

The meta model defines the structure of the application's model created by the software architect. The core elements of the meta model are depicted in Fig. 2 in form of a class diagram, because the physical representation of the model is composed of classes (currently with a subset of C#, compliant with our meta-model). This representation was chosen because it allows us to easily extend the meta model via class inheritance and to store it and work with it in its serialized form, which is itself programming-language independent, whether it be JSON, XML or YAML. During deserialization, the loaded model is automatically validated against our meta model using the class system of .NET framework, which otherwise prevents it from being instantiated.

Our meta model contains the Prototype root object, which has the following attributes:

1. Collection of 1..n applications
2. Collection of 1..n resources
3. Collection of 0..n entities

**Table 1**
Available operations.

| Operation | Input parameters | Degrees of freedom |
|---|---|---|
| **SequenceOperation**<br>Wraps multiple operations to behave as a single operation, supports multiple repetitions. | – Operations<br>– NumberOfRepetitions | – Logic to be executed<br>– Number of repetitions |
| **LoadEntitiesFromStorage**<br>Loads entities from specified entity storage in a specified quantity. | – EntityStorageName<br>– EntitySetName<br>– EntityName<br>– Filter.OnAttribute<br>– Filter.IsNominal<br>– Filter.NumberOfResults | – Storage service<br>– Type of storage<br>– Configuration and performance of the storage<br>– Size of the result set<br>– Size of the source data set<br>– Type of query |
| **InsertEntityToEntityStorage**<br>Inserts one or multiple entities into a specified entity storage. | – EntityStorageName<br>– EntitySetName<br>– EntityName<br>– NumberOfEntities | – Storage service<br>– Type of storage<br>– Configuration and performance of the storage<br>– Size of inserted data<br>– Size of already stored data<br>– Type of insert |
| **AddMessageToQueue**<br>Inserts a single message into a specified queue. Message can be processed by the Triggered action. | – QueueName<br>– EntityName | – Queue service<br>– Configuration and performance of the queue<br>– Size of inserted message<br>– Number of inserted messages |
| **ExecuteAction**<br>Remotely synchronously or asynchronously executes the defined action within the specific application in the model. | – ApplicationName<br>– ActionName | – Target application<br>– Target action<br>– Communication strategy<br>– Number of repeated executions |
| **RunComputation**<br>Run a CPU-intensive computation, which consumes a specified amount of CPU time to simulate operations with higher time complexity. | – ConsumedCPUTime | – Amount of the CPU time |

#### 4.2.1. Collection of applications

Each model must contain at least one definition of the modeled application. Multiple applications can be part of the same model, which is especially beneficial when an integration of interacting applications is to be modeled and understood.

Two relevant application types are used as a representation of the majority of cloud-deployed applications:

- ***REST API application:*** Contains a set of callable actions representing endpoints that have their own URL. HTTP request sent at defined URL triggers a defined **Action** which executes nested **Operation**.
- ***Worker application:*** Has **actions** invoked by a custom **trigger**, for example, when a message of a specific type is placed into a message queue. This application type allows us to model asynchronous processing within the prototype.

Every **Action** contains a single **Operation**, which defines what exactly should be executed. Our meta model contains multiple kinds of operations that are outlined in Table 1 and described in detail in Table B.2, Table B.3, Table B.4 and Table B.5, which further detail what kind of functionality can be modeled with these operations. These operations were selected to cover most commonly used resource-consuming operations in cloud applications. The tool is extensible (see Section 8.1.1 for details) and it is ready for new operations to be added if needed.

#### 4.2.2. Shared definitions of data entities

PaaS cloud applications are being typically designed to work with data stored in various cloud services, processed and sent to clients. To preserve model validity, we need to ensure that the modeled operations are processing realistic amounts of data, i.e. comparable to their assumed real deployment. We identified that the majority of PaaS cloud data services are working with data in some form of data **entity** (object in the key–value storage, table in the relational database, document in the document database). Thus, we model data used by the application in form of an entity, which has a set of properties of given data type. This description is independent of the implementation of the specific storage. Data entities are defined at the model level, not at the application level. The data exchange between applications can thus be modeled easily. Although we store detailed definitions of all properties of the entities for the prototype generation purposes, in the model we specify the entities in a statistically summarized form—e.g. Entity Product has 20 numeric properties and 10 string properties with an average data length of 50 characters.

#### 4.2.3. Allocated cloud resources

Every application in the PaaS cloud requires a set of allocated resources it consumes—from the execution environment where the application is hosted and can utilize its computing power, to the place where it can persist its data. We wanted to simplify and unify resource management, therefore all resources used in the model are described together with their configurations on a single place. Applications and their operations are referencing resources by their unique name. In the definition of resources it is possible to reference data entity, e.g. when we define new relational database, we define that the database has an entity set (table) for the given entity, and contains defined amount of generated records.

### 4.3. Model representation

Given the context of architecture design for our approach, object model was identified as a suitable representation for modeling the application design. Every component of the application is represented with an object and its attributes. References between objects model the dependencies between the designed components. The object model also naturally supports inheritance that enables simple extensibility. As this model representation is universal, it can be used by any object-oriented language. An instance of the model can be distributed in a JSON format—serialized form of an object in the standardized format that is well supported nowadays.

Fig. 2 shows the metamodel that was designed to fulfill the requirements identified during the analysis. One can see that the root object of the model is the Prototype. Same as an actual prototype, the object consists of a set of the applications, each having a set of operations, entities and utilized resources.

- ***Actions:*** The actions represent the functionality of the designed application. Every action encapsulates an operation that represents a concrete function that can be executed by the generated prototype.
- ***Resources:*** The operations may need to utilize allocated resources. Because of that, the Prototype has a set of resources representing cloud services. There are two basic types of resources supported at the moment: entity storages and queues. A queue allows inserting messages in order to pass data or trigger operations. An entity storage is an abstraction of every possible storage service, e.g., relational database, cache or key–value storages. The definition of the content size of an entity storage is also supported in order to increase the model precision.
- ***Entities:*** Entities represent the data model of an application. Every entity has a set of properties of a specified type and size in order to define an appropriate complexity of a data model. Entities are defined independently so that they can be shared by multiple applications, used by operations or persisted in a storage.

It is crucial for every operation, action, application, entity and resource to have its own unique name in order to prevent ambiguity during the generation process. Since resources and entities are shared by all applications, it is necessary to interconnect application and its resources and entities with their storages. Despite the possibility of inserting all resources and entities definitions into each application, these unique names are used to link them together. It is very useful from the perspective of a possible repetition in the string representation of the model.

### 4.4. Degrees of freedom in the model

Altogether the described elements of the model give us an expressive power to describe a system that consists of multiple operations that are commonly used in cloud applications, with a special focus on various storage operations, which strongly affect system performance. Different types of storage can be descried in the model (both SQL and NoSQL) utilizing various cloud services. Description of the storage also includes volume of stored data (size and number of entities) and volume of data processed by an operation. Both synchronous and asynchronous communication strategies can be used. The model describes also the execution environment for the application, controlling both vertical and horizontal scalability of the execution environment. The key degrees of freedom in the model are summarized in Fig. 3 and detailed in Tables B.2–B.5.

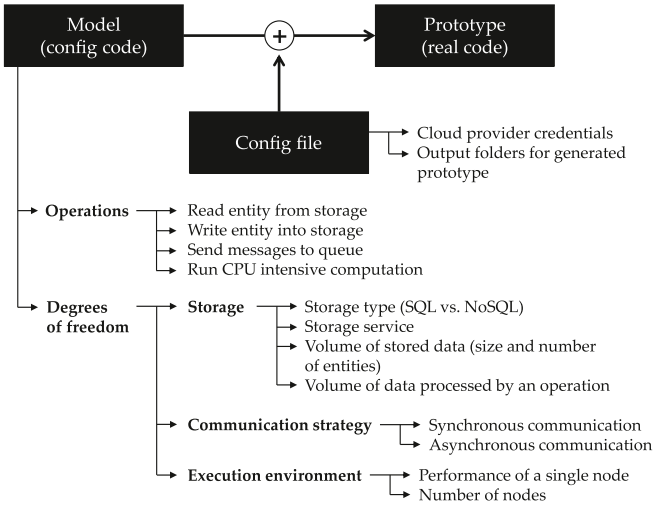


**Fig. 3.** Degrees of freedom of used application model.

```
{ "$type": "Prototype",
  "Applications": [
    { "$type": "RestApiApplication", "Name": "SimpleSampleApplication",
      "Platform": "DotNet46",
      "Actions": [
        { "$type": "CallableAction", "Name": "GetProducts",
          "Operation": {
            "$type": "LoadEntitiesFromEntityStorage",
            "EntityName": "Product", "EntitySetName": "Products",
            "EntityStorageName": "SampleDB",
            "Filter": { "$type": "FilterCondition", "UseKey": true,
              "OnAttribute": "Id", "NumberOfResults": 30 }}
        }]}],
  "Entities": [
    { "$type": "Entity", "Name": "Product",
      "Properties": [
        { "$type": "PropertyInfo", "Name": "Id", "Type": "Int32" },
        { "$type": "PropertyInfo", "Name": "Name", "Type": "String" },
        { "$type": "PropertyInfo", "Name": "Price", "Type": "Decimal" }
      ]}],
  "Resources": [
    { "$type": "AzureSQLDatabase", "PerformanceTier": "standard",
      "ServiceObjective": "S0", "Name": "SampleDB",
      "EntitySets": [
        { "$type": "EntitySet", "EntityName": "Product",
          "Name": "Products", "Count": 25000 }]},
    { "$type": "AzureAppService", "Name": "SampleWebHosting",
      "PerformanceTier": "StandardS3",
      "WithApplication": "SimpleSampleApplication" }]
}
```

**Fig. 4.** Valid simple sample model serialized in JSON format.

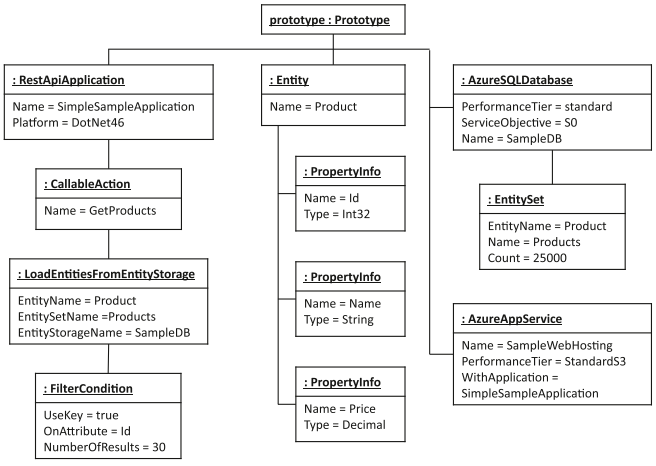


**Fig. 5.** Valid simple sample model visualized as an object diagram.

### 4.5. Model example

To better understand the structure of the models, see a simple example in Fig. 4 and 5, which models a very simple single REST API application to be deployed to Azure Web App service. This model represents a very simple .NET framework 4.6 web application using MVC pattern with a single controller and action that when invoked via its URL address loads 30 records of the Product type from a relational Azure SQL Database, which is automatically filled with 25 000 sample records in the Products table.

A more complex example of a model is in Section 7, dedicated to a case study of our approach, and further sample models can be found at project's website.[2]

### 4.6. Model validation

It is a responsibility of the user to provide the model of a desired prototype using the prototype metamodel. However, the creation of the prototype model is not trivial because one can only use a limited subset of the notation (C# in our case) corresponding to the model meta-model (and hence follow strict rules). Any ambiguity or missing information would prevent the PaaSArch Cloud Prototyper tool from the correct operation. The prototype model can only use the supported elements and has to be in the right format. These prerequisites are checked by the tool. Application modules are loaded and used to generate the executable code. If the compilation process is successful, the prototype description is valid, otherwise the user is alerted about the invalidity of the model.

### 4.7. Expressiveness

The application model we have designed fulfills requirements defined in Section 4.1 in the following way: Both common types of cloud applications (REST API and worker processes) can be modeled independently of the programming language, the prototype must be generated in a specific programming language, however it is only set as a single parameter of the application in the model. Operations used in the model are cloud provider independent, moreover operations are independent of the specific technology or paradigm (e.g. implementation of operations with data that work the same way across multiple storage resources with different storage paradigms). To maintain identical resource consumption, real cloud resources are invoked by the prototype. The model contains description of sample data and amount of data processed by executed operations. Based on this description, real data sets are generated in allocated storage services. The first option for modeling compute-intensive operations that do not interact with other cloud services is to model them by using RunComputation operation with a value of consumed CPU time parameter being estimated using micro-benchmarks of existing algorithms or using techniques described in Spinner et al. (2015). The second option is to wrap the CPU intensive algorithm into a new operation using the extensibility of the tool making its source code part of the generated prototype. Interacting applications can be modeled using the ExecuteAction operation, which remotely executes the given action in the defined application. The model and the tool offer a rich set of extensibility points, as discussed in Section 8.1.1.

[2] https://lasaris.fi.muni.cz/research/software-architecture-optimization-in-paas-cloud-applications/paasarch-cloud-prototyper-tool

## 5. Model to code transformation

The main principles of the code generation in our tool rely on the existing body of knowledge in field of Generative Programming and Model Driven Development (Czarnecki and Eisenecker, 2000). We have implemented a custom model-to-text transformation engine, which accepts our meta-model and maps elements from the meta-model on instances of transformation rules (code generators) that represent the modeled application, and its architecture. The execution of the transformation rules leads to generation of prototype's source code in a specific programming language, such as C# or Java. Our implementation is capable of resolving dependencies of generated code on multiple cloud providers (extensibility support) while working with platform abstracted operations in the model. Such a representation of application prototype in a form of code generator classes that are referencing each other based on their dependencies can be used to generate actual source code without breaking any dependencies and other related files of the prototype.

The first step in the process is the mapping of classes from the model on code generators that can generate objects of the model (e.g. specific action) in a given context. The context plays a significant role as multiple code generators can be used to generate the same operation (e.g. LoadEntityFromEntityStorage used in the sample model) but only one must be matching the operation and the context at the same time. The context is the target platform for which the prototype is generated (e.g. .NET framework 4.6), type of the application (a REST API application has slightly different architecture than a worker application) and referenced dependencies such as the cloud resources in the model. Based on the type of referenced resource, the operation must have a different implementation when it tries to access data in NoSQL database vs. in relational database, as the client libraries and access patterns are fully dependent on the used cloud service and cloud provider.

To handle resolution of all dependencies in the code generating process, a custom Dependency Injection container was implemented, which automatically loads all available code generators and for each application in the model instantiates its code generator and recursively all its dependencies based on the structure of the modeled application and given context. In the end, the model is translated into a tree of code generators, whose execution leads to the generation of all related and dependent source code files, which can be compiled into a form of an executable prototype. This fully automated process of dependency resolution supports extensibility of our tool, as new code generators can be implemented based on the provided infrastructure of classes and interfaces, and when they are compiled as part of our tool, they are automatically discovered and used.

For deeper understanding of the model to code transformation, we have included additional details, such as explanation of dependencies of key classes and interfaces used in the tool, in A. Furthermore, detailed process of the individual steps of prototype generation is depicted in Fig. 16. And to understand how the model can be extended with new types of elements, once can consult a detailed extensibility example in Section 8.4.5.

## 6. Prototype deployment and evaluation

To minimize the effort required to use the PaaSArch Cloud Prototyper tool, we have fully automated the process of resource allocation and prototype deployment. When evaluating applications using benchmarks, we consider manual data generation as a very time-demanding and tedious process, therefore we include data description in the model, and based on the model, we automatically generate all required sample data.

### 6.1. Resource management automation

It is a responsibility of the user to provide proper configuration of the tool, which includes credentials to access and manage cloud resources for cloud providers used in the model. The interface for configuration uses the straightforward key–value model where the value is returned based on the unique key. The tool expects that loaded configuration modules are able to provide information about local paths for file generation, as well as the necessary information for the communication with cloud providers. The current module uses the standard configuration file in an XML format.

Before the final prototype generation, required resources are allocated in the cloud environment, given the authentication information is correct and the resource description matches the rules. If the allocation is successful, the tool obtains needed information, for example database connection strings that can be hard-coded into the generated applications.

### 6.2. Sample data generation

In order to achieve precise approximation of real applications, the prototype must be able to utilize storage services. The inseparable part of every storage is data that must be generated by the tool in order to match the storage definition and definitions of operations using them. As it is possible to define the precise number of results on defined storage query, the data generation process must take it into consideration.

The current implementation of sample data generation uses the advantage of the code generation possibilities of the tool. Data layer library that is able to work with the prototype storage services is generated. This library also contains data model definition. After the compilation, the library is loaded by the tool. Using reflection, instances of the defined entities are created and stored into the storages using their common interface.

To achieve the precise number of results of a query, the specified attribute values are set to be unique in the precise number of instances. The operation query attributes are set to these unique values so the application can perform the query matching the exact number of entities. The rest of the entities attributes content is generated randomly with respect to their size defined in the model. To match the desired size of the storage, dummy data is generated while being checked for overlaps with the data for query operations. The rule is that the defined size of the storage must be bigger than the sum of result numbers of all query operations.

### 6.3. Prototype deployment

Generated and compiled application prototypes are uploaded into the cloud environment, into the slots prepared in resource allocation phase using the FTP protocol. At this point, the prototype is ready to be evaluated.

### 6.4. Prototype evaluation

The generated and deployed prototype is a fully functional web application publicly available and accessible from the Internet. Every Callable Action in REST API application is generated as a method of MVC controller and can be invoked using HTTP GET method call using its URL address. As the last step of the deployment process, the PaaSArch Cloud Prototyper tool provides summarized information about all endpoints of API applications so that their actions can be triggered by any tool capable of HTTP GET method call. The user can then choose from the plethora of specialized evaluation tools (Anon, 2020d,b,a) or hosted services (Anon, 2020c) for performance evaluation of web applications, providing them this set of URLs as the evaluation endpoints. Depending on the tool that is used, the user can influence the order and the intensity of the HTTP requests simulating the real load and collecting data about response times or generating maximum load to evaluate peak and sustained throughput. By changing amount of allocated resources in the cloud and repeating performance measurements, scalability of the application can be measured.

## 7. Case study

To illustrate how this approach is generally used, and to evaluate our approach and the PaaSArch Cloud Prototyper tool, this section reports on the application of the approach to one of our projects where we served as external architects of a system for collection and processing of smart home sensor data, developed by the TapHome company. Our role was centered around expert consultations related to software architecture design of the PaaS cloud service. When we were initially involved in this project, our design decisions had to be supported with a set of manually implemented prototypes, which were benchmarked to identify desired design patterns and tactics helping us to meet the quality requirements of the project. The entire prototyping process was time consuming and costly, but it was also the only way to predict the impact of our architectural decisions due to reasons explained in Section 1. In this section our goal is to repeat the prototyping and evaluation process using the introduced approach and the PaaSArch Cloud Prototyper tool for automated generation of the prototypes. Hence after introducing the project in this section, we follow the presented approach with demonstration on this case study, and at the end discuss if the tool-supported process led to the same decision that we made during the work on the real project.

### 7.1. Requirements

The designed system is responsible for collection of measurements from sensors in a smart home. Before we joined the project, the smart home was equipped with a central processing unit (the Core) installed in the building where this Core unit was responsible for collection of measurements, its real-time processing for the purpose of building control, and it was responsible for sending control commands to actuators based on activation of smart rules in the system. This system had a mobile application which was able to connect to the Core unit using web socket connection and then access short-term data stored in the Core unit.

This initial architecture of the system had two major disadvantages: first there was *high utilization of the Core unit* due to limited compute resources and local data processing, second there was only a *limited access to historical data* due to limited storage of the Core unit.

It has been decided that the designed system needs to fulfill the following functional requirements (Gesvindr et al., 2017b):

- *Measurements collection* – The system needs to be able to collect and store data from any installation of the TapHome system. The data needs to be stored remotely in the cloud, using an appropriate PaaS cloud service in Microsoft Azure, as device management and account management services are already hosted there.
- *Automated collection* – The system has to automatically initialize data collection if it is required by any active smart rule in the system.

- *Local data presence* – Collected data should be available locally for short time period so that local smart rules can work even without Internet connection.
- *Data visualization* – The collected data should be presented to the user in a mobile application, which will allow visualization of multiple metrics over various time intervals.

The architecture of the system has to be designed in the way that it meets also the following non-functional requirements:

- *Low operation costs* – The system has to take advantage of PaaS cloud services with low operation costs and has to be designed in such a way that it does not waste allocated resources.
- *High scalability* – The system needs to scale to support collection of data from more than 1 000 smart homes equipped with the TapHome Core unit. The employed cloud services should be wisely selected so that they do not impose any strict limits on the scalability of the system (e.g. poor scalability of the relational database).
- *Low utilization of the Core unit* – The system should by design lower the utilization of the Core unit as it does have very limited computational resources (700 MHz ARM11 CPU and 512 MB RAM) and should take advantage of potentially unlimited computational resources in the cloud. The current average CPU utilization of the Core unit is up to 90% due to local data processing and should be reduced below 60%.
- *High availability* – Remotely stored data needs to be accessible at least 99.9% of time.

### 7.2. Architecture design

Our goal was to design the architecture of the system in such a way that it meets all functional and non-functional requirements. The first key architectural decision was to choose whether we should take advantage of specialized IoT services available in Microsoft Azure or build our own solution based on a set of REST APIs hosted in the PaaS cloud web hosting environment. Since one of the most important requirements was to have very low operation costs, we decided to extend the current set of APIs for customer account and device management already hosted in the Azure App Service. Specialized IoT services offered by the cloud operator provide great value when an entirely new solution is created. This way the architect can build the architecture from the high variety of features available (data collection, storage, processing and device management) with fixed pricing based on the amount of processed data. But in our case we were basically extending existing solution, so we would not be able to use the rich feature set, and by using the resource collocation, we were furthermore able to host our services in currently provisioned underutilized hosting environment without any additional cost.

After this initial decision, the architecture of the web services had to be designed, which is not a trivial task, as there is a rich set of PaaS services we could use to meet the given quality requirements. Based on our experience we proposed three major architecture designs, all based on Microsoft Azure as discussed earlier, which we needed to validate prior its implementation and choose the one which adheres most to the quality requirements.

These variants were proposed based on our expert knowledge and previous experience with this platform according to quality requirements — operation costs and scalability of the solution. We were aware of limited scalability of relational databases and their higher operation costs, therefore we also wanted to design a variant utilizing cheap and highly scalable NoSQL storage. However, the NoSQL storage has limited functionality and might be less efficient with write operations, so we also evaluated asynchronous data processing as an option. In the end, the following variants of the software architecture were proposed and evaluated in terms of their scalability:

#### 7.2.1. Variant 1 — relational database with synchronous processing

The first architecture design is based on a common practice to use relational database (Azure SQL Database) as the primary storage for data of the application. Incoming measurements were processed synchronously by the web server and directly written to the relational database, which was designed in a normalized form without any manual data pre-processing (stored duplicate records in form of pre-computed aggregations). Requests to read aggregated measurements where transformed into SQL queries and were executed in the database to obtain results. Despite the fact that Azure SQL Database offers advanced features like Column Store Indexes, which would certainly improve performance of this solution (efficient data compression, very fast processing of large data aggregations) we did not use this feature as it is included only in an expensive premium performance tier.

#### 7.2.2. Variant 2 — no-SQL database with synchronous processing

The second architecture design takes an advantage of highly scalable No-SQL storage services—Azure Table Storage. This service provides inexpensive storage for large volumes of data with high throughput and low response time when a proper partitioning strategy is applied. Received measurements are written by the web server in its original form to the Azure Table Storage and at the same time pre-computed aggregations are generated and stored in different tables. It is an application of Materialized View Tactic (Gesvindr and Buhnova, 2016a), which stores precomputed data in key–value storage in such a form that the client can get results simply by loading them from a storage based on a key or range of keys without any additional data processing. This tactic had to be applied because Azure Table Storage is a key–values storage with very limited querying support which results in a need to compute aggregations by the application server not by the storage services. Without precomputed results due to a complexity of the calculations, which strongly depend on high volumes of data loaded from the storage service, the throughput of the web service that provides aggregated data would be significantly decreased. Partitioning is done based on a specific sensor, period and applied statistical function to distribute load across as many servers as possible in the data center to guarantee high scalability of this service.

#### 7.2.3. Variant 3 — no-SQL database with asynchronous processing

As we were concerned about the complexity of synchronously executed operations in Variant 2 related to the computation of measurement's aggregations immediately as new data arrive, we proposed a modified variant of the architecture introduced here. This is because the synchronously executed operations could lead to decreased throughput and limited scalability of web services for storing new measurements, leading to timeouts and lost measurements. Therefore we proposed modified variant of the architecture design utilizing the same storage service but with simplified synchronous processing of received measurements together with the application of Asynchronous Messaging Tactic (Gesvindr and Buhnova, 2016a) and Competing Consumers Tactics (Gesvindr and Buhnova, 2016a). The received measurements are stored in the Azure Table Storage without any further processing. After the measurements are stored, a message describing asynchronous request for Materialized View update is queued to a highly scalable queue service (Azure Service Bus) and processing of the request on the web server is completed. A worker process that loads messages from the queue computes and stores updated measurements aggregations without any impact on performance of web services. As in the second case, when the client loads data, no processing is necessary as aggregated measurements are stored and can be loaded with minimum effort from the storage service.

### 7.3. Model

Based on the considered architectures of the application we created three different models[3] of these architectures to evaluate the impact of different used design tactics and cloud services.

Model of the first variant is composed of 2 REST API applications:

- The first application called 'Data Collecting API' exposes a single callable action intended for storing new measurements. This calls the InsertEntityToEntityStorage operation from our library of operations, which is part of the PaaSArch Cloud Prototyper tool. The operation stores 45 instances of Measurement entity in the Azure SQL Database without any processing, which approximates a batch of measurements to be stored. This action has its own URL, which is later called by the benchmark tool.
- The second application called 'Visualization Data API' exposes a single callable action, which executes a query to the relational database to retrieve aggregated measurements. As the tool currently does not support a rich set of all possible types of queries to the database, we repeat execution of a single query, which loads a set of rows multiple times to approximate resources consumption of a complex data aggregation.

Part of the model are also the definitions of resources allocated in the cloud and utilized by the applications and their operations. In this case, the Azure SQL Database is defined and the model also contains service specific configuration (e.g. Performance Tier) which is applied during the resource provisioning phase. This database contains a single table defined as an entity set and its structure is derived from the used entity.

The last part of the model are the shared definitions of entities which are used in the application. Our goal is not to model the used entity exactly, but to approximate its size and complexity, which is determined by the number of attributes and used data types. Definition of the entity is not part of the depicted model, but we created structure which approximates received measurements in a size of 4 kB.

The second variant differs from the first by the use of different storage service. This change was easily accomplished by substitution of Azure SQL Service resource with Azure Table Storage. As this variant also synchronously updates stored aggregated values, we extended the callable action to store new measurements so that it stores raw data (same as Variant 1) but in addition also loads a portion of already stored measurements and stores aggregated data.

The third variant is the most complex one. It implements Asynchronous Messaging tactic (Gesvindr and Buhnova, 2016a) to offload resource intensive operations that does not need to be executed immediately from the web server to a worker process which can easily scale and is controlled by messages delivered through highly scalable queue service. Model of this architecture is depicted in Fig. 6 and 7. In addition to 2 REST API applications when the Data Collection API application only stores raw measurements and then adds message to the queue to schedule asynchronous update of aggregated values, this model also contains a worker application with a triggered action which is triggered by a message received trigger (we support multiple types of messages in the queue which can be bound to different actions) and executes operations which were in Variant 2 executed as part of an action that receives measurements — it loads entities from storage to be aggregated and stores aggregated values.

```
Applications": [
  { "$type": "RestApiApplication", "Name": "DataCollectionApi",
    "Platform": "DotNet46", "DeployTo": "AppHosting", "Actions": [
      { "$type": "CallableAction", "Name": "SaveRawSensorData",
        "Operation": { "$type": "SequenceOperation",
          "Operations": [
            { "$type": "InsertEntityToEntityStorage",
              "EntityName": "SerializedRawMeasurements",
              "EntitySetName": "RawMeasurements",
              "EntityStorageName": "SensorDataStorage",
              "NumberOfEntities": 1,
              "Name": "InsertRawMeasurement" },
            { "$type": "AddMessageToQueue",
              "QueueName": "MeasurementProcessingQueue",
              "EntityName": "MeasurementProcessingTask",
              "Name": "ScheduleMeasurementProcessing" }]}}]},
  { "$type": "RestApiApplication","Name": "DataVisualizationApi",
    "Platform": "DotNet46", "DeployTo": "AppHosting",
    "Actions": [
      { "$type": "CallableAction", "Name": "LoadProcessedData",
        "Operation": { "$type": "LoadEntitiesFromEntityStorage",
          "Name": "GetProcessedDataToForVisualization",
          "EntityName": "ProcessedMeasurements",
          "EntitySetName": "ProcessedMeasurements",
          "EntityStorageName": "SensorDataStorage",
          "Filter": { "UseKey": true, "OnAttribute": "ID",
            "IsNominal": false, "NumberOfResults": 1 }}}]},
  { "$type": "WorkerApplication", "Name": "MeasurementProcessingWorker",
    "Platform": "DotNet46", "DeployTo": "AppHosting",
    "Actions": [
      { "$type": "TriggeredAction", "Trigger": {
          "$type": "MessageReceivedTrigger",
          "MessageType": "MeasurementProcessingTask",
          "QueueName": "MeasurementProcessingQueue"
        },
        "Name": "BatchProcessRawData",
        "Operation": { "$type": "SequenceOperation",
          "Operations": [
            { "$type": "LoadEntitiesFromEntityStorage",
              "Name": "GetRawMeasurementsForProcessing",
              "EntityName": "SerializedRawMeasurements",
              "EntitySetName": "RawMeasurements",
              "EntityStorageName": "SensorDataStorage",
              "Filter": { "UseKey": true, "OnAttribute": "ID",
                "IsNominal": false, "NumberOfResults": 1 }},
            { "$type": "InsertEntityToEntityStorage",
              "Name": "InsertProcessedMeasurements",
              "EntityName": "ProcessedMeasurements",
              "EntitySetName": "ProcessedMeasurements",
              "EntityStorageName": "SensorDataStorage",
              "NumberOfEntities": 1 }]}}]},
```

**Fig. 6.** First part of the application model serialized in JSON format.

```
"Entities": [
  { "$type": "Entity", "Name": "SerializedRawMeasurements",
    "Properties": [
      { "$type": "PropertyInfo", "Name": "ID",
        "Type": "System.Int32", "IsIndexed": true },
      { "$type": "PropertyInfo", "Name": "SensorData",
        "Type": "System.String", "ContentSize": 11250 },
      { "$type": "PropertyInfo", "Name": "SensorData2",
        "Type": "System.String", "ContentSize": 11250 }]},
  { "$type": "Entity", "Name": "ProcessedMeasurements",
    "Properties": [
      { "$type": "PropertyInfo", "Name": "ID",
        "Type": "System.Int32", "IsIndexed": true },
      { "$type": "PropertyInfo", "Name": "ProcessedSensorData",
        "Type": "System.String", "ContentSize": 2000 }
    ]
  },
  { "$type": "Entity", "Name": "MeasurementProcessingTask",
    "Properties": [{ "$type": "PropertyInfo",
        "Name": "RawMeasurementID", "Type": "System.Int32" }]}],
"Resources": [
  { "$type": "AzureTableStorage", "Name": "SensorDataStorage",
    "EntitySets": [{ "$type": "EntitySet", "Name": "RawMeasurements",
        "EntityName": "SerializedRawMeasurements", "Count": 10000 },
      { "$type": "EntitySet", "Name": "ProcessedMeasurements",
        "EntityName": "ProcessedMeasurements", "Count": 100 }]},
  { "$type": "AzureServiceBusQueue", "Name": "MeasurementProcessingQueue",
    "SizeInGB": 2 },
  { "$type": "AzureAppService", "Name": "AppHosting",
    "PerformanceTier": "StandardS3" }]}
```

**Fig. 7.** Second part of sample application model serialized in JSON format.

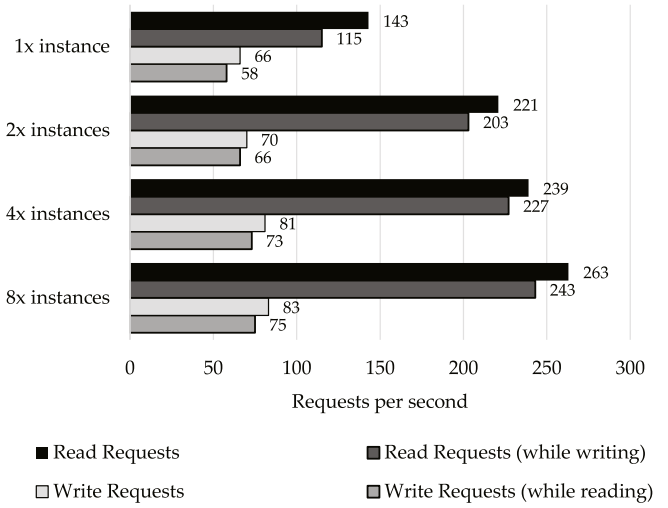


**Fig. 8.** Measured throughput of the generated prototype for Variant 1.

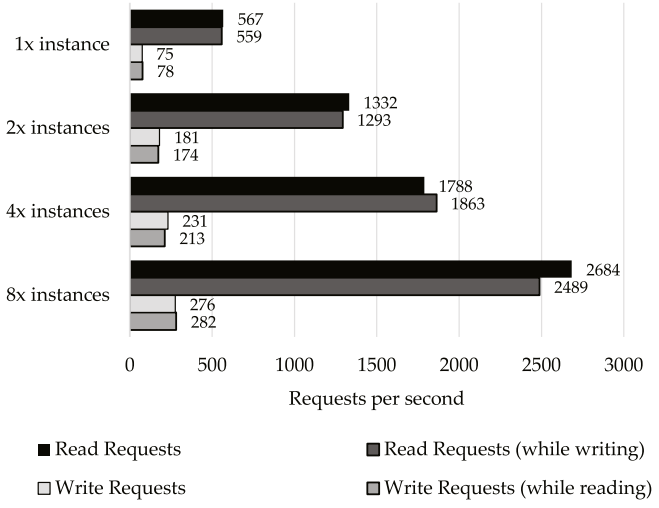


**Fig. 9.** Measured throughput of the generated prototype for Variant 2.

To sense the efficiency of our approach, it is worth mentioning that the creation of all 3 models took 60 min to an experienced software architect. The design process can be significantly accelerated, because currently the PaaSArch Cloud Prototyper tool lacks any GUI for model creation and modification, therefore in this test the architect had to compose the model in C# source code from available objects.

### 7.4. Evaluation

Based on the conducted throughput, scalability and latency measurements of the generated prototypes, we were able to evaluate the impact of our architectural decisions when designing a PaaS cloud application. Generated prototypes were benchmarked using the *wrk* HTTP benchmarking tool[4] with results presented and explained in this section.

The throughput and scalability of the first variant is depicted in Fig. 8. Azure SQL Database became very soon a performance bottleneck of the solution and despite the increased number of web server instances, the solution was poorly scalable and was not able to fully utilize allocated compute resources. In this test, we were utilizing the S3 performance tier costing 150$/month, which is not in compliance with the low operation costs requirement. We can see a decrease in the throughput for mixed workload as read and write request negatively interfere with each other overloading the database. Variant using Azure SQL Database together with asynchronous processing was not considered, as it does not improve the read throughput which was already an issue for this variant.

Throughput and scalability of the second variant (results depicted in Fig. 9) is significantly improved in comparison to the first variant as the NoSQL database has significantly better throughput and is not a performance bottleneck of the application. Still the write requests due to their synchronous processing pose a risk of timeouts when the complexity of updates of aggregated values increases.

The third variant (results depicted in Fig. 10) provides better throughput and scalability for write operations independently of their complexity due to asynchronous processing of write requests. Read throughput is the same as for the second variant.

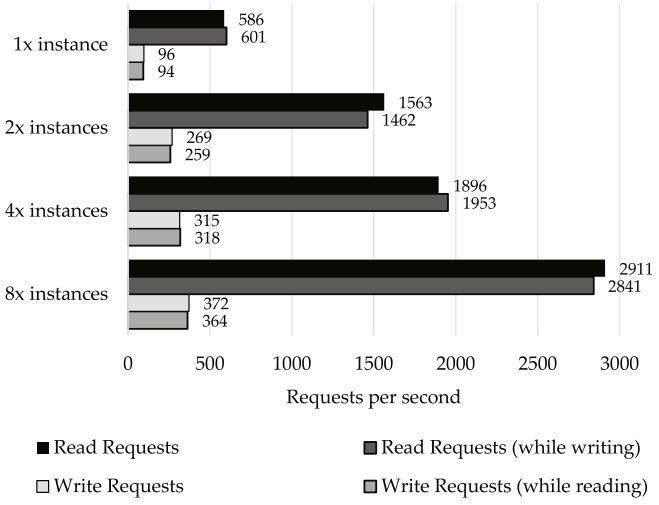


**Fig. 10.** Measured throughput of the generated prototype for Variant 3.

As part of the evaluation, we also measured and compared response latency of the deployed prototypes across all the tree variants for read and write requests. The results are depicted in Fig. 11. According to the results, the third variant provides the best response latency, as data are read from a NoSQL storage with low latency without complex query processing, and write operations only store raw data and send messages to the queue service to activate asynchronous processing of stored data. The advantage of Variant 3 in comparison to Variant 2, i.e. low latency of write operations, would increase together with the complexity of data processing, which is in case of Variant 2 done synchronously during request processing by the application server.

Based on generated prototypes and their evaluation, we were able to choose the third variant of the architecture design with the most desirable qualities based on given requirements of the project.

After the application was implemented, we conducted an additional benchmark which compares performance of the prototype for variant 3 with the final implementation of the application. Results are depicted in Fig. 12. Better performance of the final system was achieved mostly by application of Data Sharding (its impact is measured in Gesvindr et al. (2017b)) that was not part of the prototype, and other implementation details that were

[3] Source code of these models can be downloaded at project's website: https://lasaris.fi.muni.cz/research/software-architecture-optimization-in-paas-cloud-applications/paasarch-cloud-prototyper-tool

[4] https://github.com/wg/wrk

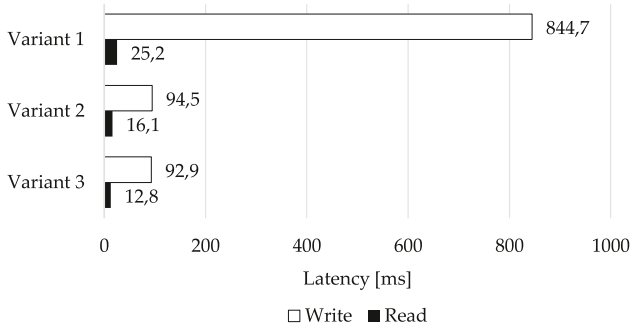


**Fig. 11.** Measured response latency of the generated prototype for all variants.

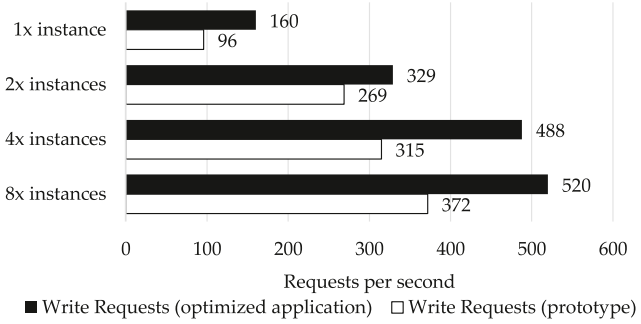


**Fig. 12.** Comparison of measured throughput of the generated prototype for variant 3 and optimized final application.

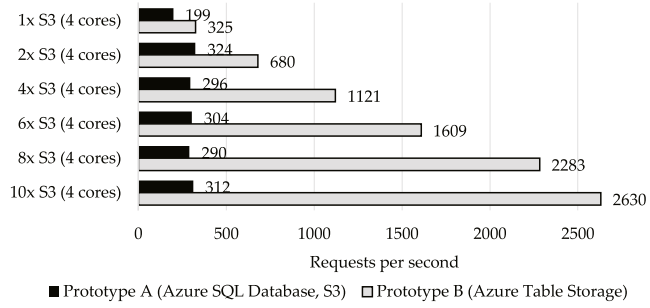


**Fig. 13.** Scalability of generated prototypes.

part of the implementation while not known during the time of prototype design. The same trend can be seen in our other case studies where the complexity of the implemented application exhibited higher level of complexity than what was assumed during design (and prototype evaluation). However, the comparative results of the prototype evaluation suggested the right combination of PaaS services, although the exact performance numbers changed from design to implementation. Results from our other case study (Gesvindr et al., 2017a) are depicted in Fig. 13 and 14. One can see that both the prototype and the final implementation has the same degree of scalability despite the fact that throughput of the prototype is higher. Evaluation of the prototype led to the decision to use an architecture utilizing Azure Table Storage which was the right decision as it performs significantly better in the implemented application too.

The sample prototype model depicted in Fig. 4 (29 lines of JSON code) generated a Visual Studio solution with 260 lines of C# code, which in this case means that for a single line of model code, almost 9 lines of C# code are generated. The serialized model in JSON has 2 KB in size, while the generated C# source code has 21.5 KB.

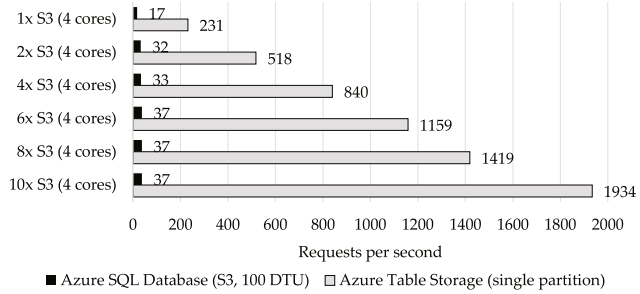


**Fig. 14.** Scalability of implemented application.

## 8. The tool

This section details the implementation of the *PaaSArch Cloud Prototyper* tool, including its non-functional characteristics, architecture, extensibility, and other implementation details.

### 8.1. Tool non-functional requirements

Since the PaaSArch cloud prototyper is meant to be further extended into a complex tool supporting PaaS application architects, we paid attention to its non-functional quality criteria. Namely the following.

#### 8.1.1. Extensibility

The extensibility was identified as the most important requirement. Over time, mainstream platforms and approaches may change, so the tool has to be designed to evolve easily as a response to these possible changes. The following three main axes were identified, where the tool should be extensible using modules implementing exposed interfaces.

- ***Applications and Platforms:*** To decide what technology should be used, the software architects may try several different programming languages or frameworks. Therefore there cannot be any restrictions from this point of view.
- ***Cloud provider support:*** As there are several cloud providers, and new ones will appear, the tool must be able to handle service management for multiple cloud providers over many different APIs. These APIs may also vary in time, or they can be replaced with completely new version. Providing the extensibility in this axis offers an easy way to experiment with services of multiple providers, choosing the best one, or split the application among multiple cloud providers.
- ***Resources and Operations:*** There is a collection of available resources and operations, from which the application results as their composition. It is important to have an option to add new ones with minimum effort and no modifications of the existing code. This is achieved by modular architecture of the tool.

#### 8.1.2. Modularity

Modularity of the system is a suitable way to support extensibility. New functionality can be added using independent modules with clearly defined interfaces and responsibilities. The inner implementation of the provided modules would determine the behavior of the whole program. Modularity also helps to increase the maintainability, where the deprecated modules can be replaced with new ones with minimum effort.

#### 8.1.3. Performance

In case of the prototype generator, the performance is not critically important unless it decreases the usability of the tool. In the context of the prototyping tool, the performance can be defined by the time needed for transformation of the model into the executable prototype. We can also measure the duration of the deployment process and sample data generation. These times are dependent on the complexity of the prototype and the allocation time of services in the cloud. Despite the fact that the performance is not the priority, the implementation should be efficient and even a complex prototype should be generated at most in a few minutes with resources provisioned at most in tens of minutes.

#### 8.1.4. Usability

The target users are supposed to be software engineering professionals with advanced knowledge of software architecture and modeling. With these prerequisites, the usage of the tool might still require some preparation and knowledge about its inner processes. As the tool functionality is completely dependent on the input by the user, the provided prototype model must be always valid. Although the usability of the tool was not the priority in the initial version of the tool discussed here, it is meant to become the priority in the future versions.

#### 8.1.5. Safety

The functionality of the tool requires access to the service management interface of a cloud provider. To automate the process, the user has to provide the access to their account. Incorrect allocation of services may cause huge costs on the user side. The tool must work with maximum care as every mistake can result in an enormous financial loss. The sensitive data provided by the user also cannot leak as it might be abused. To prevent this, user credentials are not stored in the configuration of the tool but a generated access token is used instead, which can be in case of leakage easily invalidated.

### 8.2. Tool architecture

The high-level organization of the tool is in Fig. 15. To achieve desired modularity and extensibility, the whole solution was divided into several components with clearly defined responsibilities and interfaces. The instances of these components are replaceable modules.

- ***Tool core:*** The core of the tool is responsible for the process management using other modules as instruments to complete the whole use case.
- ***Application modules:*** Application modules represent concrete implementations of application prototypes. A module of this type can extend the set of possible application types supported by the tool. The interface is not platform dependent, therefore it is possible to use multiple modules to support various platforms of the resulting application prototypes.
- ***Prototype generator:*** As every application can be represented as a set of files, the universal generator is sufficient to transform the object representation of the application into the executable prototype.
- ***Build providers:*** Some platforms require build process to transform code files into the executable application. A build module instance shall be present for each supported platform to perform this process.
- ***Configuration providers:*** The configuration with user settings is obtained via the interface of configuration provider module. This information is used during the generation process, deployment process or to customize the tool behavior. Based on the implementation, the configuration can be obtained from locally stored files or by more sophisticated ways.
- ***Deployment providers:*** The support of multiple cloud providers is enabled by these modules. A module of this type handles the interaction between the tool and a specific cloud provider. The responsibilities of these modules are to allocate and deallocate resources utilized by generated prototypes. Also, deployment providers are necessary to deploy generated prototypes to the cloud environment.
- ***Benchmark modules:*** The benchmark modules provide support during the evaluation process. Based on concrete implementation they provide functionality to enable the quality measurement of the generated prototypes.
- ***Sample data generators:*** As the user does not have the exact specification of the data model in the design phase, it is necessary to generate sample data based on the general description of entities to be saved in storages to meet requirements described in the application model. The process of the sample data generation is determined by the implementation of this module.

### 8.3. Implementation

The prototyping tool was developed based on a previous analysis with respect to the introduced architecture design to achieve required functionality and quality attributes. The .NET framework with C# as a programming language were chosen as a suitable platform for this kind of project. This platform is widely spread and well supported, with extensive community of programmers and rich tooling support.

The whole implementation places emphasis on the replaceability and extensibility. The tool is organized into modules in the form of .NET code libraries that are loosely coupled with the core of the tool via defined interfaces. The core manages the prototyping process using these modules as instruments. The workflow of prototype processing is depicted in Fig. 16. The modules are loaded at runtime based on the phase of the prototyping process. This design allows us to replace modules with a different implementation or to extend prototyping options adding new ones.

The current implementation of the tool supports generation of .NET/C# applications (Console and Web API) that can use several Microsoft Azure services (Azure Table Storage, Azure SQL Database, Azure Service Bus and others). At the same time, the tool is prepared for future extensions and support of prototype generation for other platforms and cloud providers.

### 8.4. Extensibility

The tool is prepared for several levels of extensions. Basically, every new module can replace the old one or extend existing options of the tool. Modules can provide build process for multiple platforms, multiple ways to generate sample data or different approaches to evaluation support.

However, the most common extensions are expected to add new functionality to the output applications. The essential means for extensibility is via abstract model inheritance and implementation of proper interfaces. The idea is that a built library will be placed into the tools folder. While parsing the given application model, the tool finds the library during the scanning of its base folder for types that can satisfy model requirements.

The extensibility of the tool is supported on multiple levels:

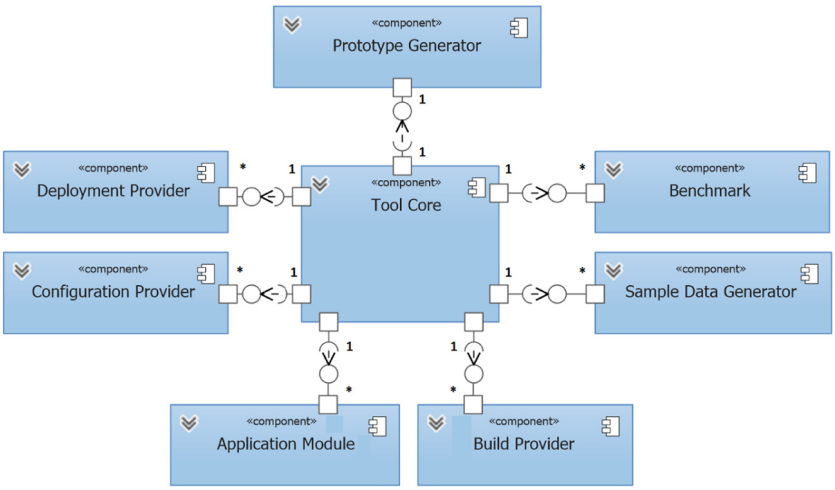


**Fig. 15.** Tool architecture.

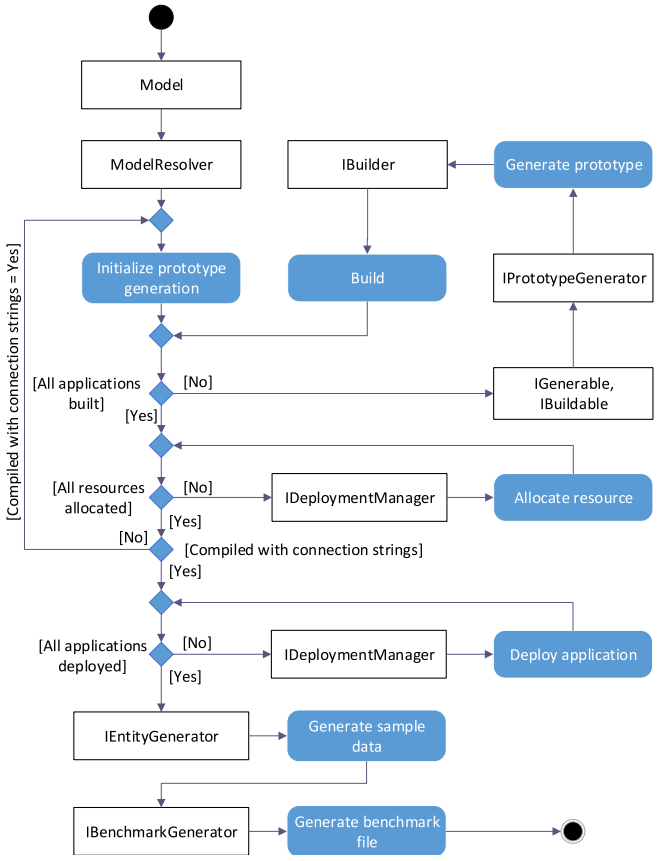


**Fig. 16.** Workflow of prototype processing.

### 8.4.1. Types of applications

The tool currently provides a basic library of reusable code generators. Adding new application type support (something else than REST API application and worker application) includes inheriting a new class from the Application base class and an implementation of the GeneratorManager<T>abstract class. While the prototype model is being resolved, the resolver is iterating over all applications in the prototype finding the GeneratorManager class with matching generic parameter T that represents application type having the platform identification property. The responsibility of this class is to assemble the code generators to complete an executable application and handle the specifics of the chosen application type.

### 8.4.2. Platforms and programming languages

The interface for application modules abstracts the rest of the tool from the concrete implementation of the in-memory application representation and the programming language paradigm. Adding support for prototypes generated on other platforms and using different programming languages includes creation of new text transformation templates for a specific language and their encapsulation within new code generator classes.

### 8.4.3. Cloud providers

The support of multiple cloud providers is a desirable feature of the tool. The first step when adding new cloud provider support is inheriting resource classes from the prototype model and adding provider-specific properties to them. Then, it is required to create code generators that are able to utilize these modeled resources. The last step is an implementation of deployment provider module that handles the communication with a new cloud provider and is responsible for allocation of newly added cloud resources.

### 8.4.4. Operations and resources

To add support for new resources, such as a relational database or a message bus, one has to inherit a proper base class from the abstract prototype model and describe resource properties. It is the same for operations. Every operation model has to inherit from Operation base class. The operation-specific parameters have to be added so they can be used in templates.

Implementing resource or operation support for the .NET platform involves creating generator classes that inherit from the Modelable<T>base class, where the T parameter represents a class from the application model. When the model is being resolved, the resolver scans all types and searches for a match in the generic parameter in all resources and operations. All such classes are instantiated and together with their dependencies they are generated as executable code as part of the prototype.

Resource extensibility is more complex, as it also requires the implementation of resource management using cloud provider API so that the tool can automatically allocate and deallocate the resource. Different cloud providers have different resource management API, but this API is usually unified for all their resources. Therefore the resource management logic can be reused when implementing a new resource of the currently supported provider.

```
public class CallUrl : Operation
{
    public string Url { get; set; }

    public override List<ResourceReference> GetReferencedResources()
        => new List<ResourceReference>();
    public override List<string> GetReferencedEntities()
        => new List<string>();
}
```

**Fig. 17.** CallUrl class.

```
public class CallUrlGenerator : Modeled<CallUrl>, IOperation
{
    public OperationInterfaceGenerator OperationInterface { get; set; }

    public CallUrlGenerator(string projectName,
        OperationInterfaceGenerator operationInterface,
        CallUrl modelParameters, bool canInitialize = true)
        : base(projectName, "Operations", modelParameters.Name,
              typeof(CallUrlOperationTemplate), modelParameters,
              modelParameters.Name, canInitialize)
    {
        OperationInterface = operationInterface;
    }
}
```

**Fig. 18.** CallUrlGenerator class.

### 8.4.5. Operations extensibility sample

To illustrate the extensibility of the tool, we describe an implementation of a new operation called *CallUrl*, which sends a HTTP GET request to a specified URL and downloads the response. To this end, the following files need to be implemented:

1. *CallUrl class* – Class that is used in the model as an operation and stores parameters of the operation (called URL). The source code is presented in Fig. 17.
2. *CallUrlGenerator class* – This class specifies how the model class is converted into the source code. It specifies what code-generating template is used, and how the parameters and context are passed to the template. The source code is presented in Fig. 18.
3. *CallUrlTemplate template* – This is a text-processing template that contains generated code with template parameters that are replaced with values passed from the generator. The source code is presented in Fig. 19.
4. *CallUrlRegistrations class* – It registers the code generator class into a Dependency Injection container to correctly resolve all dependencies during model processing. The source code is presented in Fig. 20.

## 9. Discussion

*Application context.* This approach was designed to enable quality evaluation of most common PaaS cloud applications, which are web applications hosted using PaaS cloud services with a logic executed on the backend server frequently communicating with other cloud services (storage, queue, etc.). We support generation of two different types of applications. i.e. REST API applications and worker applications. The REST API applications are common backend for modern JavaScript-based web applications and mobile applications. Despite the fact that we do not support rendering of HTML pages using a specific UI framework in our generated prototype, the REST API application can substitute this type of application as the internal logic being executed is very similar. Only the output is formatted using different framework. To support asynchronous communication between multiple components of the application, a worker application which processes messages from the queue with a custom logic is also supported. Our approach is however not intended for prototyping of desktop applications and evaluation of their UI behavior and performance.

```
<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ parameter
    type="CloudPrototyper.NET.Framework.v462.Computing.Generators.CallUrlGenerator
    "
    name="Model" #>
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Net.Http;

namespace  <#= Model.Namespace #>
{
    public class <#= Model.Name #>
        : <#= Model.OperationInterface.Namespace #>.<#=
          Model.OperationInterface.Name #>
    {
        public const string Key = "<#= Model.ModelParameters.Name #>";

        public void Execute(List<string> outputs)
        {
            string result;
            var request = WebRequest.Create("<#= Model.ModelParameters.Url #>");
            request.Method = "GET";

            var response = request.GetResponseAsync().Result;
            using (var reader = new StreamReader(response.GetResponseStream()))
            {
                result = reader.ReadToEnd();
            }
            outputs.Add(result);
        }
    }
}
```

**Fig. 19.** CallUrlTemplate template.

```
public class CallUrlRegistrations
    : GeneratorDependency<CallUrlGenerator>
{
    public override List<IRegistration> GetRegistrations(string projectName)
        => new List<IRegistration>
    {
        Component.For<OperationInterfaceGenerator>()
            .ImplementedBy<OperationInterfaceGenerator>()
            .LifestyleSingleton()
            .DependsOn(Dependency.OnValue("projectName", projectName))
    };
}
```

**Fig. 20.** CallUrlRegistrations class.

*Limitations to effectiveness.* The biggest limitation to effectiveness of our approach is the limited set of operations and resources supported by our model. To create this set, we identified the most common cloud resources and operations that were present in the applications we consulted in the past and implemented them as part of the model in the current version of the tool. But as cloud providers are heavily investing in an extension of their PaaS cloud offerings, new services are still emerging, which prior to the first use in the model need to be implemented using extensibility of our model and the tool which leads to additional effort to sustain good effectiveness of our approach. But when the model is extended, these extensions can be shared and the whole community using the tool can benefit from them.

*Approach scalability.* Neither the approach or tool itself impose any limitations in the model size. The only possible bottleneck is the implementation of the code generator, which in case of performance issues can be optimized and improved for proper handling of extremely large models. However, models of realistic sizes are currently generated in seconds, compiled in tens of seconds, and the cloud resources are allocated in minutes or tens of minutes based on the number of allocated resources (which can be allocated in parallel if needed).

*Model reusability.* Elements of the model, such as Applications, their Actions, and parts of their logic in the form of Operations can be easily copy-pasted to a different model (in the same representation — JSON/XML/source code), same as Resources and

Entities. During the copy-pasting, especially in case of Operations, one needs to be careful to include correct references to the Resources and Entities, when referring via their names. The naming of the elements in the model may be challenging, as names of elements must be unique; otherwise, they could not be indisputably referenced.

*Language and runtime performance impact.* Despite the fact that the tool itself is implemented in C# and .NET framework, the template framework we use was selected so that it can produce code in any programming language. The current implementation of the tool generates code of the prototype in C# and .NET framework. Measured performance of the prototype is affected by the language and framework used by the prototype. But the impact of the actual programming language and framework on the performance of the application is very small, comparing to the impact of the performance, scalability and delays in the communication with cloud resources used by the prototype.

*Adoption with Agile and DevOps methodologies.* The approach can be very well integrated within agile methodologies, which can use this approach within the design phase of each iteration/increment or sprint to evaluate proposed changes in the software architecture of the implemented application, to manage stable fulfillment of performance requirements during the development process. The use of our tool can be automated as a part of the CI/CD pipeline — models can be stored in source control, downloaded to the build agent together with the tool, which is then executed as a command-line utility. A set of performance tests can be executed automatically after a successful deployment of the prototype.

*Simplification of application design.* As the model of the application reflects limited knowledge of the application, available during early design time when the architecture design is taking place, multiple implementation details are better to be mimicked in the model via more general constructs rather than modeled in detail (Becker et al., 2009). To abstract from the details of the implementation concepts without depreciation of quality evaluation results, it is important to identify and model the most significant operations in terms of *processing delay* (e.g. a blocking remote call to the used cloud service or long-running blocking CPU operation) or *potential throughput and scalability limitation* (e.g. limited or unknown throughput of the used cloud service). This process, as known also from other model-driven performance prediction research (Becker et al., 2009), takes an advantage of expert knowledge in combination with simulation and micro-benchmarking. E.g., for CPU intensive operations or unsupported cloud services, the processing delay can be simulated by RunComputation operation with parameters estimated based on micro-benchmarks of the algorithm or service.

## 10. Conclusion

The Platform as a Service (PaaS) cloud computing model introduces enormous potential for software architects who are able to leverage it. In our experience, however, it is very difficult for software architects to navigate the complexity of the PaaS cloud model. Thus the architects either end up combining PaaS services randomly (with high cost and inefficiency), or adopt the same practices they were using to in the on-premise environment (with many bottlenecks, e.g. in terms of a relational database).

In our work, we aim at creating a tool set assisting software architects in cloud application design that will be effectively utilizing the PaaS cloud platform and combining the available services in an optimal fashion. In this paper, we take advantage of the cloud elasticity and introduce a design-time quality evaluation technique for PaaS applications based on automatically generated application prototypes, which can be deployed to the cloud and evaluated in the context of multiple quality attributes and environment configurations. As part of this work, we discuss the implementation of the approach in terms of the PaaSArch Cloud Prototyper tool, which automates the approach for the software architect. The whole approach was demonstrated on a case study of a real IoT application.

In the future, we will follow the extensibility goals introduced in this paper, and further improve the whole approach and its tool set. Moreover, we want to use the tool set to evaluate architectural patterns for PaaS cloud application design, which we are currently formulating.

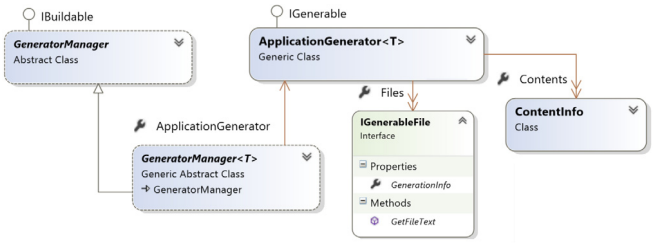


**Fig. A.21.** Generator managers class diagram.

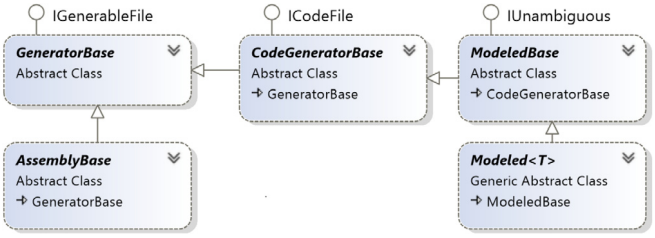


**Fig. A.22.** Generator class diagram.

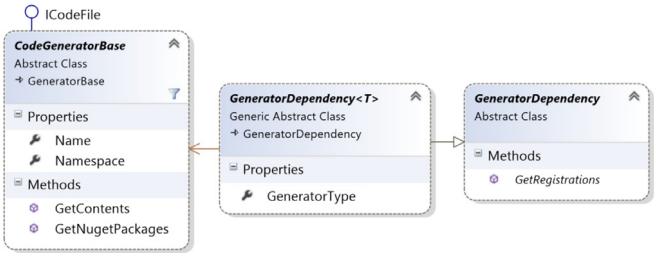


**Fig. A.23.** Generator dependencies model.

## CRediT authorship contribution statement

**David Gesvindr:** Conceptualization, Methodology, Writing - original draft, Visualization, Validation. **Ondrej Gasior:** Software, Investigation, Data curation. **Barbora Buhnova:** Supervision, Writing - review & editing.

## Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jss.2020.110701.

## Acknowledgment

This research was supported by ERDF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No. CZ.02.1.01/0.0/0.0/16_019/0000822).

**Table B.2**
Available operations — Part 1.

| Operation | Input parameters | Degrees of freedom |
|---|---|---|
| **SequenceOperation** — Wraps multiple operations to behave as a single operation, because a root level action can contain only a single operation. This operation is similar to a block command in programming languages. | **Operations** – List of operations to be executed<br>**NumberOfRepetitions** — All operations can be executed multiple times, e.g. simulation of iterative entity processing. | – **Logic to be executed.**<br>– **Number of repetitions** to efficiently describe resource intensive pattern when entities are processed iteratively which leads to high resource consumption of utilized resources due to multiple invocations. |
| **LoadEntitiesFromStorage** — Loads entities from specified entity storage in a specified quantity. The tool automatically generates sample data in such a distribution and generates matching condition so that only a required amount of entities is retrieved from the storage. This operation is completely independent of used storage service, it can be used the same way when working with a relational database, NoSQL key–value storage or a document database. The generated prototype will have a correctly functioning platform and storage-specific code generated. | **EntityStorageName** — Reference to the used storage defined as a resource of the prototype.<br>**EntitySetName** — Reference to an entity set defined as part of a referenced storage (this can be table, document collection depending on the type of the used storage service; but this operation works the same way across multiple storage technologies).<br>**EntityName** — Type of the entity stored in the defined entity set. Entities are defined globally in the model, so that they can be shared across applications.<br>**Filter.OnAttribute** — Specifies which attribute of the entity is used for filtering purposes.<br>**Filter.IsNominal** — Specifies whether generated values and filter will work with nominal values or ordinal values (which leads to a range query in the storage)<br>**Filter.NumberOfResults** — Specifies how many entities should be retrieved from the storage. This attribute influences distribution of the generated values in an attribute of the entity used for filtering. | – **Storage service** used to store and retrieve data<br>– **Type of storage** — SQL, different NoSQL storages; depends on used storage service<br>– **Configuration and performance of the storage** — Specific storage service resource exposes platform specific configuration of the cloud resource including configuration of allocated performance<br>– **Size of the result set** = Number of results x Entity size. Number of results is configured as part of the filter, which influences data generation process and generates adequate filter condition for the query. Entity size depends on the number and type of attributes for the given entity defined in the model.<br>– **Size of the source data set** = Number of entities in the entity set x Entity size. Number of stored entities in the entity set is a configuration of the entity set specified in the model.<br>– **Type of query** – If the values are generated as nominal, exact match (equality) queries will be generated, if filter is specified as ordinal, range queries will be used instead. |

**Table B.3**
Available operations — Part 2.

| Operation | Input parameters | Degrees of freedom |
|---|---|---|
| **InsertEntityToEntityStorage** — Inserts one or multiple entities into a specified entity storage. This operation is completely independent of the used storage service, it can be used the same way when working with a relational database, or NoSQL key–value storage, or a document database. The generated prototype will have a correctly functioning platform and storage specific code generated. | **EntityStorageName** — Reference to the used storage defined as a resource of the prototype<br>**EntitySetName** — Reference to an entity set defined as a part of referenced storage into which the entities will be inserted.<br>**EntityName** — Type of entity stored in defined entity set. Entity is defined globally in the model, so that they can be shared across applications. Values of entity attributes are generated randomly.<br>**NumberOfEntities** — The number of entities that will be inserted into the storage. By using this property, multiple entities can be inserted in a single batch. | – **Storage service** used to store and retrieve data.<br>– **Type of storage** — SQL, different NoSQL storages; depends on the referenced storage service.<br>– **Configuration and performance of the storage** — Specific storage service resource exposes platform-specific configuration of the cloud resource including configuration of the allocated performance.<br>– **Size of inserted data** = Number of the inserted entities x Entity size. Number of the inserted entities is configured as part of the operation. Entity size depends on the number and data type of the attributes for a given entity defined in the model.<br>– **Size of already stored data** = Number of entities in the entity set x Entity size. The number of stored entities in the entity set is a configuration of the entity set specified in the model.<br>– **Type of insert** — Insert of a single entity or batch insert can be executed. If multiple entities are to be inserted using isolated inserts, then a single entity insert operation should be nested in a sequence operation with multiple repetitions specified. If Number of Entities to insert is set to a higher number, batch insert can be used, if supported by the storage service. |

**Table B.4**
Available operations — Part 3.

| Operation | Input parameters | Degrees of freedom |
|---|---|---|
| **AddMessageToQueue** — Inserts a single message into a specified queue. Triggered action in the same or different application can listen to the same queue and receive the inserted message to start an asynchronous execution of the action. This operation is completely independent of the used queue service. | **QueueName** — Reference to a used queue service defined as a resource of the prototype.<br>**EntityName** — Type of an entity inserted to the queue. Size of the message depends on the size of the serialized entity, which is instantiated with random values in its attributes. | – **Queue service** used to receive the message.<br>– **Configuration and performance of the queue** – Depending on the cloud-provider specific queue, services expose configuration of allocated performance and queue partitioning, which can be then configured in our model.<br>– **Size of inserted message** depends on the number and data type of attributes for a given entity defined in the model.<br>– **Number of inserted messages** can be influenced by nesting this operation in a sequence operation with multiple repetitions specified. |
| **ExecuteAction** — Remotely executes defined action in defined application in the model. The primary purpose of this operation is to allow modeling of microservice architecture, where the application is decomposed into a multiple microservices (in our model defined as additional applications) that communicate frequently with each other via service calls. | **ApplicationName** — Reference to an application defined in the model.<br>**ActionName** — Reference to an action defined in the referenced application that should be executed. | – **Target application** can be the same application or different application in the model independently of where the application is hosted. This can be used for the design of microservice architecture or cross-cloud applications.<br>– **Target action** is the action specified on the target application that will be executed<br>– **Communication strategy** depends on type of the target action — if it is a callable action (has a public URL generated after deployment), the execution strategy is a synchronous call. If the target action is a triggered action with a queue trigger, then the asynchronous pattern is used and the invocation is automatically done by sending a message of the correct type to a queue service bound to the trigger. Necessary implementation details (service URL, queue) are automatically derived by the tool from the model.<br>– **Number of repeated executions** can be influenced by nesting this operation in a sequence operation with multiple repetitions specified. |

**Table B.5**
Available operations — Part 4.

| Operation | Input parameters | Degrees of freedom |
|---|---|---|
| **RunComputation** — Run a CPU-intensive computation, which consumes a specified amount of CPU time to simulate operations with higher computational complexity (e.g. photo resampling) running isolated at the application server without any communication to other cloud services, which could affect the overall application server utilization and delay request. | **ConsumedCPUTime** — The amount of CPU time in milliseconds that must be consumed by the executed computation before this operation is completed. | – **Amount of the CPU time** that is consumed. |

## Appendix A. Model-to-code transformation

In this section, additional details about the implementation of the model to code transformation in our approach are given. Dependencies of the key classes and interfaces used in the tool are explained. The diagram of the important classes is depicted in Fig. A.21. Every application module is an implementation of the GeneratorManager abstract class. Using generator managers, every application defined in the application model is transformed into a set of files as required by the IGenerable interface. The generic T parameter represents a specific type of Application object from the prototype model.

Text files (source code, project files) are represented by so-called *generators*. These classes must implement the IGenerableFile interface that prescribes the GenerationInfo property having information useful during the generation and the GetFileText method that returns the text of the future text file.

Since the interface for generated text files is general in order to support every possible programming language, several levels of inheritance have to be present in order to support .NET implementations. The created class hierarchy is depicted in Fig. A.22. An abstract GeneratorBase class implementing the IGenerableFile interface is the root of the hierarchy. In the .NET solution, there are basically only two types of text files. Firstly, there are plain text files: solution file, project files or configuration files. These generators will inherit right from the base class with exception

of assembly file generator that has also the AssemblyBase class in order to better support the project file specific format. The second type of files is source code files. Every code file generator inherits from the CodeGeneratorBase class that implements code file interface that prescribes Name and Namespace properties. Such a code file represents one class in the resulting project.

Although most of the generators are fixed to create the infrastructure of the final application, there are also generators representing the actual parts of the model. Every instance of Entity, Resource, Action and Operation class has its own generator implementation that inherits from the Modeled class. Such an instance can be unambiguously identified using a unique name of the T parameter that represents one of mentioned model parts. The results classes can encapsulate the code working with resources, a logic of operations, or definition of data entities.

In the phase of prototype generation, relations between the future classes are represented by relations of their generators. If a future class A is dependent on the future class B, the generator of class A has a reference to the generator of class B. The code is generated using runtime T4 templates. These precompiled templates consist of the text and expression blocks. Expression blocks are replaced with the values of the model of the template at runtime and the template can produce text of any text file. The model of the template is the class generator itself so its properties can be used in the templates expression blocks. References in the generated classes are possible due to Name and Namespace properties of ICodeFile of referenced generators. The information from the prototype model can be accessed via an instance of T in the Modeled<T> class.

Within a generator manager instance of the application, an IoC container is used. Based on the application model, a generator manager registers step by step all the required generators. IoC container resolves all generator instances that implement the IGenerable interface. This results in a complete set of text files that can be generated. To ensure that all dependencies can be satisfied during the dependency resolution process, the implementation of GeneratorDependency class, shown in Fig. A.23, is created. The GetRegistrations method returns IoC container registrations of all referenced generators. This implementation supports extensibility and ensures that all required files will be present in the generated prototype.

However, not all dependencies are generated by the PaaSArch Cloud Prototyper tool itself. The resulting files can reference libraries distributed via NuGet manager. If this is needed, it is required to implement the GetNugetPackages method to return the list of desired packages. During the generation process, references to these packages are added to project files. Before the compilation, these packages are downloaded to the solution folder. The support of NuGet package manager is crucial because it is usually the only way to get libraries necessary for the work with cloud services.

It is similar to the binary files that are referenced from the code. The GetContents method shall return the list of locations of these files. In the generation phase, the information about the relative path to the file is used to determine the output path. The information shall correspond to the path used in the file template.

## Appendix B. Available operations

This section consists of Table B.2, Table B.3, Table B.4 and Table B.5, with detailed description of what kind of functionality can be modeled with the operations available in the prototype models.

## References


Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I., 2013. Software architecture optimization methods: A systematic literature review. IEEE Trans. Softw. Eng. 39 (5).

Anon, 2016. The Total Economic Impact of Microsoft Azure Platform-As-A-Service. https://azure.microsoft.com/en-us/resources/total-economic-impact-of-microsoft-azure-paas.

Anon, 2017a. ASP.NET Dynamic data. https://msdn.microsoft.com/en-us/library/ee845452.aspx. (Accessed 12 January 2017).

Anon, 2017b. Cloud products & services - Amazon web services (AWS). https://aws.amazon.com/products/. (Accessed 24 April 2017).

Anon, 2017c. Microsoft azure: Cloud computing platform & services. https://azure.microsoft.com/en-us/. (Accessed 24 April 2017).

Anon, 2020a. Ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. (Accessed 08 January 2020).

Anon, 2020b. Apache JMeter. https://jmeter.apache.org/. (Accessed 08 January 2020).

Anon, 2020c. Run URL-based load tests with Azure DevOps. https://docs.microsoft.com/en-us/azure/devops/test/load-test/get-started-simple-cloud-load-test?view=azure-devops. (Accessed 08 January 2020).

Anon, 2020d. Wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. (Accessed 08 January 2020).

Barbierato, E., Gribaudo, M., Iacono, M., Jakóbik, A., 2019. Exploiting cloudsim in a multiformalism modeling approach for cloud based systems. Simul. Model. Pract. Theory (ISSN: 1569-190X) 93, 133–147, Modeling and Simulation of Cloud Computing and Big Data. http://dx.doi.org/10.1016/j.simpat.2018.09.018 http://www.sciencedirect.com/science/article/pii/S1569190X18301436.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice. Addison-Wesley, ISBN: 9780321154958.

Becker, S., 2008. Coupled Model Transformations for Qos Enabled Component-Based Software Design (Ph.D. thesis). Universität Oldenburg.

Becker, S., Koziolek, H., Reussner, R., 2009. The palladio component model for model-driven performance prediction. J. Syst. Softw. 82 (1).

Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2019. Microservices in industry: Insights into technologies, characteristics, and software quality. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 187–195. http://dx.doi.org/10.1109/ICSA-C.2019.00041.

Brataas, G., Stav, E., Lehrig, S., Becker, S., Kopčak, G., Huljenic, D., 2013. Cloudscale: Scalability management for cloud systems. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering. ICPE '13, ACM.

Buyya, R., Ranjan, R., Calheiros, R., 2009. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In: High Performance Computing Simulation, 2009. HPCS '09. International Conference on. pp. 1–11. http://dx.doi.org/10.1109/HPCSIM.2009.5192685.

Chen, F., Grundy, J., Schneider, J.-G., Yang, Y., He, Q., 2015. Stresscloud: A tool for analysing performance and energy consumption of cloud applications. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2. ICSE '15, IEEE Press.

Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications. Addison Wesley, ISBN: 9780201309775, https://books.google.cz/books?id=4CPmr3qcVvYC.

Erl, T., Puttini, R., Mahmood, Z., 2013. Cloud Computing: Concepts, Technology & Architecture, first ed. Prentice Hall Press, Upper Saddle River, NJ, USA.

Ferry, N., Chauvel, F., Song, H., Rossini, A., Lushpenko, M., Solberg, A., 2018. CloudMF: Model-driven management of multi-cloud applications. ACM Trans. Internet Technol. (ISSN: 1533-5399) 18 (2), 16:1–16:24.

Fowler, M., 2002. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN: 0321127420.

Franceschelli, D., Ardagna, D., Ciavotta, M., Di Nitto, E., 2013. SPACE4CLOUD: A tool for system performance and costevaluation of cloud systems. In: Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds. ACM.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN: 0-201-63361-2.

Gesvindr, D., Buhnova, B., 2016a. Architectural tactics for the design of efficient PaaS cloud applications, in: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2016.

Gesvindr, D., Buhnova, B., 2016b. Performance challenges, current bad practices, and hints in PaaS cloud application design. SIGMETRICS Perform. Eval. Rev. 43 (4).

Gesvindr, D., Buhnova, B., 2019. PaaSArch: Quality evaluation tool for PaaS cloud applications using generated prototypes. In: 2019 IEEE International Conference on Software Architecture, Companion Proceedings (ICSA). IEEE, pp. 170–173.

Gesvindr, D., Buhnova, B., Gasior, O., 2017a. Quality evaluation of PaaS cloud application design using generated prototypes. In: 2017 IEEE International Conference on Software Architecture (ICSA). pp. 31–40. http://dx.doi.org/10.1109/ICSA.2017.43.

Gesvindr, D., Michalkova, J., Buhnova, B., 2017b. System for collection and processing of smart home sensor data. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 247–250. http://dx.doi.org/10.1109/ICSAW.2017.23.

Gorton, I., 2006. Essential Software Architecture. Springer Science & Business Media.

Homer, A., Sharp, J., Brader, L., Narumoto, M., Swanson, T., 2014. Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications. Microsoft patterns & practices.

Klaussner, C., Lehrig, S., (2014). Using Java EE ProtoCom for SAP HANA cloud. In: SOSP'14 Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days 2014, p. 17.

Koziolek, H., 2010. Performance evaluation of component-based software systems: A survey. Perform. Eval. 67, Special Issue on Software and Performance.

Lehrig, S., Eikerling, H., Becker, S., 2015. Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In: Proceedings of the 11th Int. ACM SIGSOFT Conference on Quality of Software Architectures. ACM, ISBN: 978-1-4503-3470-9, pp. 83–92.

McGrath, G., Short, J., Ennis, S., Judson, B., Brenner, P., 2016. Cloud event programming paradigms: Applications and analysis. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD). pp. 400–406. http://dx.doi.org/10.1109/CLOUD.2016.0060.

Pahl, C., Jamshidi, P., Zimmermann, O., 2018. Architectural principles for cloud software. ACM Trans. Internet Technol. 18 (2), 17.

Parra, C., Cleve, A., Blanc, X., Duchien, L., 2010. Feature-based composition of software architectures. In: Babar, M.A., Gorton, I. (Eds.), Software Architecture. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-642-15114-9, pp. 230–245.

Pérez, A., Moltó, G., Caballer, M., Calatrava, A., 2018. Serverless computing for container-based architectures. Future Gener. Comput. Syst. (ISSN: 0167-739X) 83, 50–59. http://dx.doi.org/10.1016/j.future.2018.01.022, http://www.sciencedirect.com/science/article/pii/S0167739X17316485.

Spinner, S., Casale, G., Brosig, F., Kounev, S., 2015. Evaluating approaches to resource demand estimation. Perform. Eval. (ISSN: 0166-5316) 92, 51–71. http://dx.doi.org/10.1016/j.peva.2015.07.005, http://www.sciencedirect.com/science/article/pii/S0166531615000711.

Taylor, R.N., Medvidovic, N., Dashofy, E.M., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley Publishing.

van Eyk, E., Toader, L., Talluri, S., Versluis, L., U, A., Iosup, A., 2018. Serverless is more: From PaaS to present cloud computing. IEEE Internet Comput. (ISSN: 1941-0131) 22 (5), 8–17. http://dx.doi.org/10.1109/MIC.2018.053681358.

Wilder, B., 2012. Cloud Architecture Patterns, first ed. O'Reilly Media, ISBN: 978-1-4493-1977-9.

**David Gesvindr** is a Ph.D. student at Masaryk University (MU), Faculty of Informatics in Brno, studying the design of effective PaaS cloud applications as his main research focus within the Lab of Software Architectures and Information Systems (Lasaris). He has been awarded as Microsoft Most Valuable Professional, in Microsoft Azure and in Data Platform specialization. He has participated in numerous industrial cloud projects and is a leading organizer of the Czech Windows User Group, the largest Czech Microsoft professional community.

**Ondrej Gasior** contributed to the work during his master studies at the Faculty of Informatics, Masaryk University, where he recently graduated from the Applied Informatics study program, studying the architecture design of cloud applications as his main research focus within the Lab of Software Architectures and Information Systems (Lasaris) at the faculty. Now he works in industry as .NET/C# Software Developer at Barclays Investment Bank.

**Barbora Buhnova** is an Associate Professor and vice-dean for industrial partners at Masaryk University (MU), Faculty of Informatics in Brno. She received her Ph.D. degree in computer science in 2008, and continued as a postdoc researcher at University of Karlsruhe and Research Center for Information Technology (FZI) in Karlsruhe, Germany, and later at Swinburne University of Technology in Melbourne, Australia. She is the Steering Committee chair of the ICSA conference (CORE rank A) and has been involved in organization of numerous leading conferences (e.g. ICSE, ICSA, ESEC-FSE, QoSA, CBSE, MOBILESoft). She acts as a reviewer and guest editor in multiple journals (e.g. IEEE TSE, Elsevier SCP, Elsevier JSS, Springer SoSyM, Wiley SME), and is member of the IEEE TSE Review Board.