Towards Omnia: a Monitoring Factory for Quality-Aware DevOps

Marco Miglierina
Politecnico di Milano
Via Golgi 42, 20133
Milan, Italy
marco.miglierina@polimi.it

Damian A. Tamburri
Politecnico di Milano
Via Golgi 42, 20133
Milan, Italy
damianandrew.tamburri@polimi.it

ABSTRACT

Modern DevOps pipelines entail extreme automation and speed as paramount assets for continuous application improvement. Likewise, monitoring is required to assess the quality of service and user-experience such that applications can continuously evolve towards use-centric excellence. In this scenario however, it is increasingly difficult to pull up and maintain efficient monitoring infrastructures which are *frictionless*, *i.e.*, they do not introduce any slowdown neither in the DevOps pipeline nor in the DevOps organizational and social structure comprising multiple roles and responsibilities. Using an experimental prototype, this paper elaborates *Omnia* an approach for structured monitoring configuration and rollout based around a monitoring factory, *i.e.*, a re-interpretation of the factory design-pattern for building and managing ad-hoc monitoring platforms. Comparing with practitioner surveys and the state of the art, we observed that Omnia shows the promise of delivering an effective solution that tackles the steep learning curve and entry costs needed to embrace cloud monitoring and monitoring-based DevOps continuous improvement.

Keywords

Monitoring, Monitoring Management, Monitoring Factory, Monitoring Interface, Monitoring Infrastructure as Code, Monitoring Configuration as Code

1. INTRODUCTION

The advent of cloud computing triggered a huge change in software release cycles for an increasing number of companies embracing cloud technologies as the 21st century's technological utility. Upfront investments in physical servers are being replaced by on-demand and pay-per-use cloud access while complex manual deployment procedures are automated in the context of DevOps [7]. However, where automation is needed to speedily deploy new versions of a service in a safe and reproducible way, monitoring is required to assess the quality of service and the user experience in order to understand and solve problems and take business decisions equally fast. Also, both automation and monitoring play multiple roles in a complex organizational and social structure [19] around the cloud application, its required middleware (often from an open-source community) and indoor components.

In our previous work [13] we observed that effective monitoring in such a complex organizational structure and ecosystem [18] is still a difficult task, hardly affordable by small and medium enterprises where resources and expertise availability are scarce. Although a huge number of monitoring tools, both commercial and open-source ones, proliferated in the last few years, there is no holistic framework that drives the embracing of standardized monitoring solution [6] for monitoring while big corporations with high expertise such as Google, Facebook or Netflix are developing and refining their own appropriate solutions.

The main objective of this work is to provide an initial investigation of such a standardized monitoring solution, by offering an approach called *Omnia*, whose key objective is reducing the learning curve and entry-cost to monitoring technologies. Omnia is an approach that assists system administrators in deploying a monitoring system and developers in configuring and accessing monitoring information, exploiting DevOps practices such as Infrastructure as Code and automation [20]. Omnia consists of two major parts: (1) a *monitoring interface* for developers that helps using monitoring systems, independently of the specific implementation, and (2) a *monitoring factory* for system administrators that helps building a monitoring system that is compatible with such interface, leveraging existing monitoring tools.

Our approach is a reinterpretation of the factory pattern [10]. Similarly to the famous design pattern, our monitoring factory creates a concrete implementation of a monitoring system (by automatically composing and configuring existing monitoring tools) and users refer to it using a common monitoring interface that is independent of the actual implementation.

Together with the implementation of the tool, we also propose the definition of a common reference vocabulary for resources being monitored and general purpose metrics, versioned with the Omnia source code and to which every component should adhere when integrated with our tool. Although many research works exist in the scope of monitoring research (*e.g.*, as highlighted by several surveys in the field [9, 6]), a fundamental lack in the monitoring research scenario is the definition of a reference dictionary as for monitoring is concerned. On the one hand, the approach

ICPE '17 Companion, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4899-7/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3053600.3053629

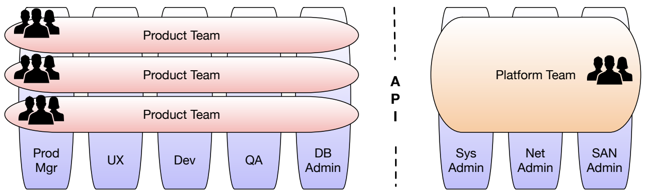

Figure 1: Team organization at Netflix. Retailored from [12]

proposed in this paper helps system administrators address the multitude of available tools and easily setup a monitoring system [13]. On the other hand, Omnia assists all software practitioners throughout all phases of their monitoring infrastructure lifecycle (*e.g.*, dashboard configuration, data exchange, analytics representation and more) providing to the entire organizational structure a single protocol, a common vocabulary and a versionable monitoring configuration language, compatible with any monitoring system deployed via the Omnia monitoring factory.

In conclusion, comparing the proposed research solution with challenges and pitfalls observed in industrial practice [13, 17], we argue that although in a prototype stage, Omnia and connected technical contributions offer a valuable basis to enter the complex and often (very) expensive world of monitoring infrastructures for cloud applications.

The rest of the paper is structured as follows. Section 2 outlines the organizational and socio-technical scenario that Omnia was designed to address as well as required terminology and motivations for this contribution. Section 3 describes the approach and the technological contributions. Section 4 compares the state of the art with our solution. Section 5 concludes the paper.

2. RESEARCH PLAYGROUND

This section outlines the organizational and socio-technical scenario that Omnia was designed to address. More in particular, we elaborate on the domain assumptions and terminology typical of the scenario we have in mind. Even though the approach could be extended to different usage scenarios, for the design of an initial prototype we consider a scenario in which a cloud application is structured according to the microservices architecture pattern along with the typical organizational-social structure [19] connected to that pattern - the scenario we address is tailored from the one adopted at Netflix [12] (Figure 1).

2.1 Domain Assumptions

Omnia assumes that each *product team* is responsible for its own product (or service), which is implemented as a microservice, for which source-code is maintained in a separate versioned repository and following an organization where development and deployment cycles are still independent from each other. Conversely, the *platform team* is cross-functional: it is in charge of supporting product teams providing infrastructure support, e.g., via APIs, orchestration software, middleware and similar technology. Such platforms are either managed by a public cloud provider, managed in-house, or a mix of these two. However, according to Netflix, there are at least three key properties that shall define the platform usage: "API-driven, self-service and automatable" [12].

2.2 Terminology

This section recaps the terminology and common vocabulary we use throughout this paper:

- **Metric**: a measurable property of a phenomenon that can be quantitatively determined. Example: response time is a metric measuring the "elapsed time between the end of an inquiry or demand on a computer system and the beginning of a response" [11].
- **Monitoring datum**: a single measurement of a metric. Example: the authentication service took 100 ms to respond.
- **Resource**: anything that can be monitored and, consequently, the source of a monitoring datum. Example: a web server, a database, a virtual machine, a container.
- **Data collector**: a software component in charge of collecting monitoring data from a resource, also called *monitoring agent*.

2.3 Motivations

Standardizing a way to describe what every product team would like to see and be notified about is definitely challenging, since every monitoring tool has its own peculiarity and is usually focused on delivering value from a specific perspective. For example, a graphing tool maybe able to plot multiple time series on the same graph for simplifying the comparison, or some analysis tool may be able to compute prediction or perform statistics that another tool is not able to perform. Or else, some tool may be able to send app notifications while another is only able to send emails.

With Omnia, our goal is to find a reasonable subset of standard features a company with small to medium cloud resources (*e.g.*, personnel, expertise, consultancy, budget or otherwise) would like to have available, from a monitoring perspective. Omnia assumes that, stemming from these standard features, that very same company can gradually and incrementally: (a) add new features to its own indoor monitoring "language"; (b) push monitoring tool vendors to implement the missing ones or alternatively, (c) elaborate further on their own monitoring (micro)services to fulfill new feature requests.

In this scenario, every product team can describe its Monitoring Configuration as Code, and keep that code versioned together with its services code in the root of its repository. The next section elaborates further on this key idea, which constitutes the basis of the Omnia approach.

3. THE OMNIA APPROACH

Considering an organizational and social structure like the one we described in Section 2.1, we describe how monitoring and its management is addressed today and how it can be addressed with our approach. Figure 2 depicts a fictional scenario where a company with low budget constraints wants to monitor its microservices architecture using existing open source monitoring tools. After studying existing solution, the platform team decides to build a monitoring system composed of 3 different components on the server

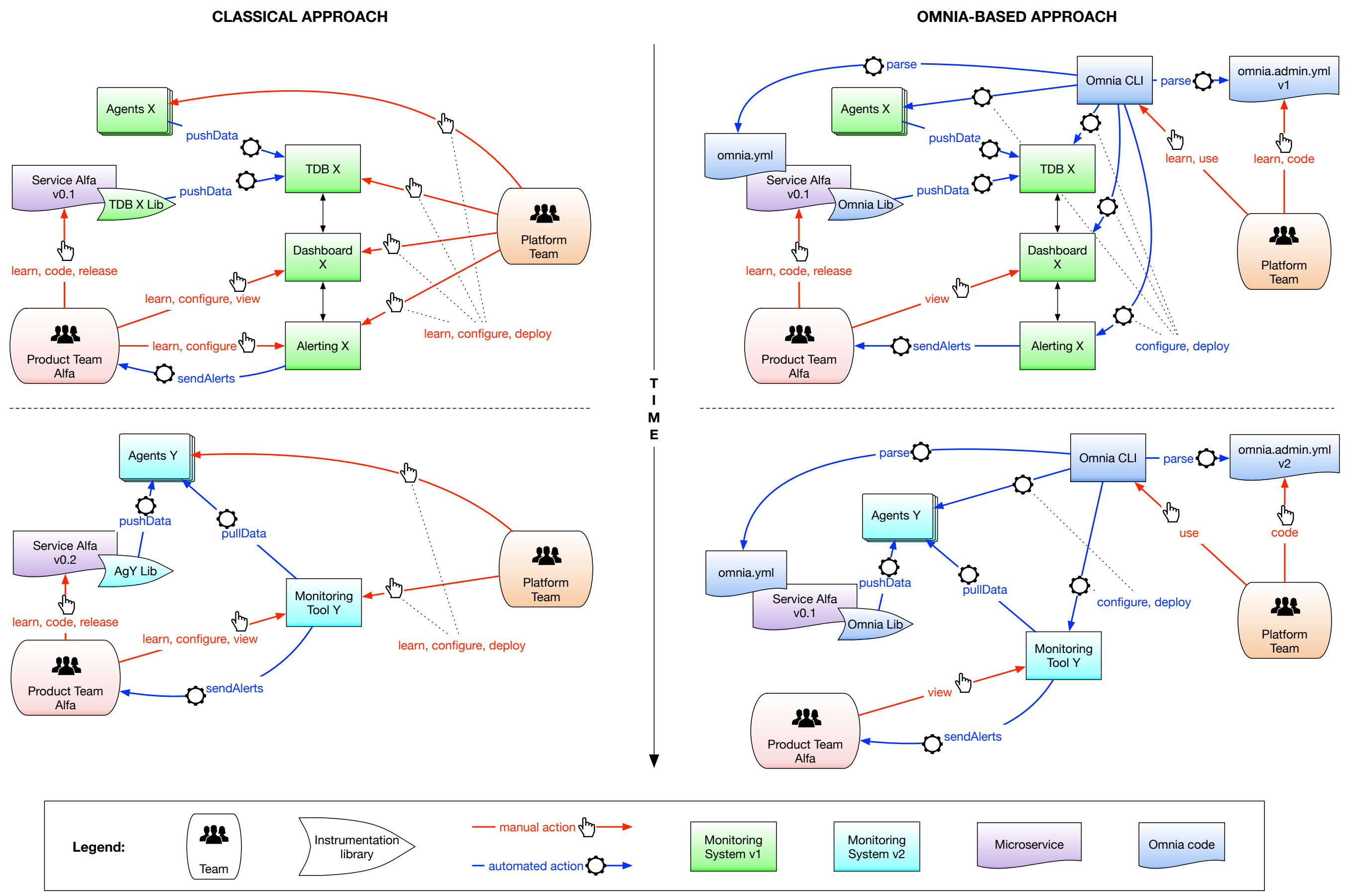

Figure 2: Comparison between the classical approach and the Omnia approach when adopting different monitoring solutions.

side: the temporal database *TDB X* for storing historical data, the *Dashboard X* for exposing time-series in form of graphs via a web interface, and *Alerting X* for sending notifications to product teams. Moreover, the team decides to adopt *Agent X* as data collectors to be run as daemons on the hosts.

On the left of Figure 2 the reader can look at how the adoption would work according to the classical approach. The platform team has to learn how to use the different tools, configure and deploy them. The platform team would then ask product teams to instrument their microservice with a vendor dependent instrumentation library, *i.e.*, the *TDB X Lib*. Product teams have to learn how to configure and use the graphical user interfaces provided by the deployed monitoring tools in order to setup their graphs and alerts. The entire process require a steep learning curve and most of the work is manual or can be automated using custom scripts.

By using Omnia, on the other hand, most of the process is automated. The platform team has to describe the system using the proposed Infrastructure as Code approach. The automated setup and deployment is carried out transparently by the *Omnia CLI*, using a convention over configuration approach. Product teams describe their graphs and alerts using the proposed Configuration as Code approach and keep the file versioned in their code base.

After few months of practical experience with the installed monitoring system, performance problems as well as usability issues are raised and the platform team decides to switch to a more simple all-in-one monitoring solution, offering storing, graphing and alerting features in a single application and using a pull strategy for retrieving data instead of having agents pushing data to the time series database (bottom side of Figure 2). On the left side, we can see how the migration process would work in a classical scenario. Most of the effort carried out by all teams is thrown away, and an additional learning step is required. The platform team has to configure the new platform and deploy it. Product teams has to release a new version of the microservice with a new instrumentation library, *i.e.*, *AgY Lib*, has to learn how to use the new graphical interface of *Monitoring Tool Y* and reconfigure all required graphs and alerts.

By using Omnia, on the other hand, no additional learning step is required. Product teams are not even required to touch their code. They just start using the new dashboard, with the same kind of graphs they defined for the first version of the platform already available. Alerts will be received as well as configured in the previously released Configuration as Code file. The platform team only requires to update the Infrastructure as Code file and trigger a new deployment phase via the *Omnia CLI*.

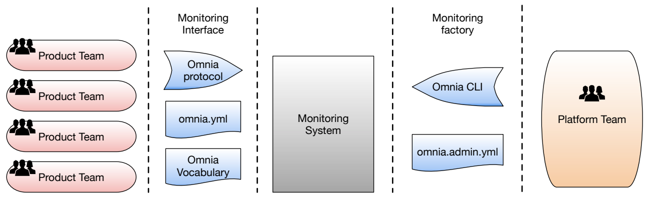

Figure 3: Omnia technological contributions

Both the product- and the platform-team workflows can be reiterated multiple times independently.

In the following sections we are going to detail the technological contributions we overviewed in the above example scenario. Figure 3 depicts such contributions and highlights how such decoupling between teams is obtained.

3.1 The monitoring interface

3.1.1 The Omnia vocabulary

The *Omnia vocabulary* is a dictionary of terms defining naming conventions for resource types and metrics that are common to all applications. It is supposed to be extended and maintained together with Omnia development and the addition of tools and libraries. Whenever Omnia is extended for supporting a new collecting tool, such tool should be adapted to agree with Omnia vocabulary. If some metric or resource is missing, this should be added to the Omnia vocabulary. The same is valid when building instrumentation libraries. Users can obviously specify custom metrics, but meta-data such as the application name should be added to monitoring data, possibly in a transparent way, using terms from the Omnia vocabulary.

Here is a first version of the Omnia vocabulary with some examples of resources definitions:

Resource	**Description**
host	a physical or virtual machine
service	5 an application
service_id	a unique identifier for an instance of an application
container_id	a unique identifier for a Linux container
container_image	a Linux container image

Metrics can be categorized according to the resource being monitored. For example, here is a short list for host and Java metrics:

Host metrics	**Java metrics**
cpu_usage_user	*heap_memory_usage*
cpu_usage_system	*thread_count*
cpu_usage_idle	*loaded_class_count*
mem_used	*garbage_collection_time*
mem_used_percent	*thread_count*

Both files are maintained together with Omnia source code.

3.1.2 The Omnia protocol and instrumentation libraries

The Omnia protocol specifies how monitoring data should be serialized and sent by data collectors to the other tools composing the monitoring system. We decided to adopt an existent and widely adopted protocol, that supported multi-dimensional meta-data in form of key values, since we could rely on existing community provided libraries and ease the adoption of industries. Such protocol is the Statsd protocol, with Influxdb tagging extension[1].

Once a protocol is defined, it is important to maintain instrumentation libraries that adhere to such protocol and enforce Omnia conventions. There exist an ever growing number of languages and frameworks, and each combination of these requires a library. A Omnia-compatible instrumentation library must adhere to the following mandatory requirements:

MR1. Use the Omnia protocol, *i.e.*, the Influx Statsd protocol to serialize metrics and send metrics	
MR2. Use the common Omnia vocabulary (Section 3.1.1) for decorating metrics with meta-data	
MR3. Set http://collector:8125 as default endpoint for sending metrics	

Also it should adhere to the following optional requirements:

OR1. Require the least possible instrumentation effort and overhead to product teams
OR2. Offer an API that is not supposed to change in the near future
OR3. Favor convention over configuration, for example by automatically inferring meta-data to be added to monitoring data

For a first prototype, a Java library for the Spring Cloud framework[2] was developed[3]. Instrumentation only requires to add a Maven[4] dependency to the project and decorate the main application class with the *@EnableOmnia* annotation. This would automatically enables the collection of default Spring Boot Actuator metrics (*e.g.*, heap memory usage or thread count) to the default endpoint (*i.e.*, http://collector:8125) and the addition to all metrics of the *service* and *service_id* meta-data. Moreover, developers can easily describe additional custom metrics, such as the number of payments processed by the service instance, using the the API provided by the Spring Boot Actuator library as described in the following example.

```
@Service
public class MyService {
    private final CounterService counter;

    @Autowired
    public MyService(CounterService counter) {
        this.counter = counter;
    }

    public void pay() {
        this.counter.increment("payments");
    }
}
```

Listing 1: Custom metrics instrumentation

[1]https://www.influxdata.com/getting-started-with-sending-statsd-metrics-to-telegraf-influxdb/
[2]http://projects.spring.io/spring-cloud/
[3]https://github.com/mmiglier/omnia-spring-boot
[4]https://maven.apache.org

In the prototype developed for this work, additional system level meta-data such as the host name where the service is running, will be added to each metric by the data collector (or agent).

3.1.3 Monitoring Configuration as Code

The *omnia.yml* file is a versionable YAML file used for configuring the monitoring activity, such as metrics time series to be plotted or alerts. Whenever a new version of the *omnia.yml* file is pushed, Omnia will update the product team specific dashboard together with any alert specified in the document. If this configuration file was standard and shared among tools, the monitoring team would be free to update the tool set of monitoring tools without interfering with the product teams work. During a first phase, interpreters and translators should be provided to compile the standard configuration file into the tool specific configuration format, with the hope an increase of popularity of the standard format, it may become widely adopted.

The first version of the *omnia.yml* is composed of two sections: (1) the *dashboard* section, where things to be visualized are described, and (2) the *action* section, where things to be done in response to events are configured. The dashboard can be composed of different kind of graphs, such as time series or pie charts. In the action section, product teams can describe actions such as email or SMS notifications, but could also adaptive actions to be triggered.

An example of *omnia.yml* file is shown in the following listing:

```
dashboard:
  timeseries:
    - metric: payments
      compute: rate
    - metric: java_heap_memory
    - metric: cpu
      compute: average by host
    - metric: ram
actions:
  email:
    - condition: http_errors / http_requests >
        0.1
```

Listing 2: omnia.yml file for monitoring configuration.

The file shall be automatically validated during automatic integration tests. Also, the file shall favor convention over configuration: every missing piece of information shall be configured using standardizable defaults.

3.2 The monitoring factory

3.2.1 Monitoring Infrastructure as Code

The *omnia.admin.yml* file depicted in Figure 3 is a file where the monitoring system is described as Infrastructure as Code and product teams repositories are listed.

This Monitoring Infrastructure as Code file is composed of 3 parts (1) the *provisioner*, where the platform team is supposed to specify the provisioner the *Omnia CLI* should use to provision the monitoring system, (2) the *tools* section, where the platform team has to list the monitoring tools, the functionalities they offer and their interconnections, and (3) the *team_repos* section, where product teams repositories are listed.

For our first prototype we experimented Omnia integrations with the following popular tools: CollectD, Telegraf, InfluxDB, Prometheus, Graphite, Grafana, Riemann. Each one offer different configuration languages and different functionalities. The user is responsible of setting what role should cover each tool under the *provides* field: agent, dashboard or actions.

We here provide an example of *omnia.admin.yml* file:

```
provisioner:
  name: docker
  args:
    username: mmiglier
    images_tag: latest
tools:
  telegraf:
    provides:
        - agent
    pushes_to:
      - influxdb
  influxdb:
  grafana:
    pulls_from:
      - influxdb
    provides:
        - dashboard
        - actions
teams_repos:
  - "github.com/mmiglier/omnia-examples/
      service1"
  - "github.com/mmiglier/omnia-examples/
      service2"
  - "github.com/mmiglier/omnia-examples/
      service3"
```

Listing 3: An example of omnia.admin.yml file.

3.2.2 The Omnia CLI

The *Omnia CLI* is the application that is used by the platform team to deploy a monitoring system that implements the *monitoring interface* according to the *omnia.admin.yml* (see Figure 3). The application exposes three simple commands:

- ***compile***, the CLI parses the *omnia.admin.yml* file, retrieves *omnia.yml* files from team repositories and creates the required configuration files required by the chosen provisioner to deploy the monitoring system;
- ***deploy***: the CLI deploys the monitoring system using the API offered by the chosen provisioner;
- ***stop***: the CLI stops the monitoring system using the API offered by the chosen provisioner.

The platform is written in Go[5] and is easily extensible with new provisioners and new monitoring tools by using the Go *template package*[6]. During compilation time, both provisioner's and tools' configuration files are generated from templates by applying to them a data structure generated from the *omnia.admin.yml* and the *omnia.yml* file. Besides templates for configuration, a developer extending *Omnia CLI* with a new tool has to create a *setup.sh* and a *run.sh* script, which will be executed for setting up the tool and running it respectively.

Starting from the Monitoring Infrastructure as Code example in Listing 3, the *Omnia CLI* will generate the code required to provision the monitoring platform according to the chosen provisioner. In our first prototype only the Docker

[5]https://golang.org
[6]https://golang.org/pkg/text/template/

provisioner was implemented and the final compilation will be a Docker Compose file together with required scripts and configuration files.

A prototype implementation of this component has been released on GitHub[7].

4. RELATED WORK

Existing monitoring solutions, such as Grafana [2] or InspectIT [4], usually provide graphical configuration interfaces as default option. For tools where a configuration language is available, it is always a custom DSL and often highly detailed [3]. Monitoring Configuration as Code approaches have been used by using existing configuration and management tools such as Puppet [5] or Chef [1] with custom plugins for writing checks [8, 16]. However this approach is not portable across multiple monitoring platform and only provide checks and not dashboards configuration.

Regarding the standardization of a common exchange format attempts have been done both in the academia [14] and in the industry [15], however there is yet no strong adoption of such proposals. Our approach does not aim at creating a new protocol or a new vocabulary, we rather aim at reducing product teams efforts, preventing them to care about the underlying protocol and make metrics available to them self-service. As for the underlying protocol, we aim at reusing existing contributions and de-facto standards.

5. CONCLUSION AND FUTURE WORK

In this paper we proposed Omnia, an approach and a tool with the key objective of reducing the learning curve and entry-cost to monitoring technologies. The concept of *monitoring factory* was introduced, as a reinterpretation of the famous design pattern where the concrete implementation of a monitoring system is kept hidden to developers via a common monitoring interface. Deployment and configuration of the monitoring platform is automated via the simple API offered by the monitoring factory we presented. System administrators can leverage the proposed Monitoring Infrastructure as Code to easily compose the set of existing monitoring tools to use and configure their roles and interconnections.

Our monitoring interface allows to separate the development workflow of the core application from the monitoring system, increasing agility and reduce effort required. The monitoring factory permits to switch across different monitoring solutions, automating the deployment of a solution that is compliant with our monitoring interface.

The proposal we offer in this paper is a baseline approach for further experimentation of wrapping monitoring solutions within a common abstraction layer. To our knowledge, this is the first attempt and it is exposed to the risk of simplifying tools specific characteristics. In this paper we addressed very simple example for demonstration purposes. We planned to address real world examples where monitoring requirements have been selected by the teams according to real world scenarios and better evaluate our approach.

6. ACKNOWLEDGMENTS

The authors' work is partially supported by the European Commission grant no. 644869 (H2020 - Call 1), DICE

[7]https://github.com/mmiglier/omnia

7. REFERENCES

[1] Chef. https://www.chef.io. Accessed: 2017-02-17.

[2] Grafana. http://grafana.org. Accessed: 2017-02-17.

[3] Grafana scripted dashboard. http://docs.grafana.org/reference/scripting/. Accessed: 2017-02-17.

[4] Inspectit. http://www.inspectit.rocks. Accessed: 2017-02-17.

[5] Puppet. https://puppet.com. Accessed: 2017-02-17.

[6] G. Aceto, A. Botta, W. de Donato, and A. PescapÃČÂÍ. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[7] L. J. Bass, I. M. Weber, and L. Zhu. *DevOps - A Software Architect's Perspective*. SEI series in software engineering. Addison-Wesley, 2015.

[8] K. Buytaert. Monitoring in an infrastructure as code age. PuppetConf, 2013.

[9] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *J. Parallel Distrib. Comput.*, 74(10):2918–2933, 2014.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object oriented software, 1995.

[11] IBM. *IBM Dictionary of Computing*. McGraw-Hill, Inc., New York, NY, USA, 10th edition, 1993.

[12] T. Mauro. Adopting microservices at netflix: Lessons for team and process design. https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/, 2015 (accessed January 16, 2017).

[13] M. Miglierina. *Monitoring Modern Distributed Software Applications: Challenges And Soloutions.* PhD thesis, Politecnico di Milano, 2017. Under revision.

[14] D. Okanović, A. van Hoorn, C. Heger, A. Wert, and S. Siegl. *Towards Performance Tooling Interoperability: An Open Format for Representing Execution Traces*, pages 94–108. Springer International Publishing, Cham, 2016.

[15] D. Plaetinck. Metrics 2.0: An emerging set of conventions, standards and concepts around timeseries metrics metadata. http://metrics20.org.

[16] S. Porter. Infrastructure as code & monitoring. AutomaCon, 2015.

[17] R. Rabiser, M. Vierhauser, and P. GrÃijnbacher. Assessing the usefulness of a requirements monitoring tool: a study involving industrial software engineers. In L. K. Dillon, W. Visser, and L. Williams, editors, *ICSE (Companion Volume)*, pages 122–131. ACM, 2016.

[18] S. SchÃijtz, T. Kude, and K. Popp. The impact of software-as-a-service on software ecosystems. In *4th International Conference on Software Business*, Potsdam, Germany, 2013.

[19] D. A. Tamburri, P. Lago, and H. van Vliet. Organizational social structures for software engineering. *ACM Comput. Surv.*, 46(1):3, 2013.

[20] L. Zhu, L. Bass, and G. Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.