Continuous Architecting of Stream-Based Systems

Marcello M. Bersani, Francesco Marconi and Damian A. Tamburri
Politecnico di Milano
Milan, Italy
Email: [$marcellomaria.bersani$, $francesco.marconi$, $damianandrew.tamburri$]@polimi.it

Pooyan Jamshidi, Andrea Nodari
Imperial College London
London, UK
Email: [$p.jamshidi$, $a.nodari15$]@imperial.ac.uk

***Abstract*—Big data architectures have been gaining momentum in recent years. For instance, Twitter uses stream processing frameworks like Storm to analyse billions of tweets per minute and learn the trending topics. However, architectures that process big data involve many different components interconnected via semantically different connectors making it a difficult task for software architects to refactor the initial designs. As an aid to designers and developers, we developed OSTIA (On-the-fly Static Topology Inference Analysis) that allows: (a) visualising big data architectures for the purpose of design-time refactoring while maintaining constraints that would only be evaluated at later stages such as deployment and run-time; (b) detecting the occurrence of common anti-patterns across big data architectures; (c) exploiting software verification techniques on the elicited architectural models. This paper illustrates OSTIA and evaluates its uses and benefits on three industrial-scale case studies.**

I. Introduction

Big data applications process large amounts of data for the purpose of gaining key business intelligence through complex analytics using machine-learning techniques [1], [2]. These applications are receiving increased attention in the last years given their ability to yield competitive advantage by direct investigation of user needs and trends hidden in the enormous quantities of data produced daily by the average Internet user. According to Gartner [3] business intelligence and analytics applications will remain a top focus for Chief-Information Officers (CIOs) of most Fortune 500 companies until at least 2017-2018. However, the cost of ownership of the systems that process big data analytics are high due to high infrastructure costs, steep learning curves for the different frameworks (such as Apache Storm [4], Apache Spark [5] or Apache Hadoop [6]) involved in designing and developing big data applications and complexities in large-scale architectures and their governance within networked organizations.

In our experience with designing and developing big data architectures, we observed that a key complexity lies in quickly and continuously evaluating the effectiveness of such architectures. Effectiveness, in big data terms, means that the architecture as well as the architecting processes and tools are able to support design, deployment, operation, refactoring and subsequent (re-)deployment of architectures continuously and consistently with runtime restrictions imposed by big data development frameworks. Storm, for example, requires the processing elements to represent a Directed-Acyclic-Graph (DAG). In toy topologies (comprising few components), such constraints can be effectively checked manually, however, when the number of components in such architectures increases to real-life industrial scale architectures, it is enormously difficult to verify even these "simple" structural DAG constraints. We argue that the above notion of architecture and architecting effectiveness can be maintained through continuous architecting of big data applications consistently with a DevOps organisational structure [7], [8]. In the big data domain, continuous architecting means supporting the continuous and incremental improvement of big data architectural designs - e.g., by changing the topological arrangement of architecture elements or any of their properties such as queue lengths - using a constant stream of analyses on running applications as well as platform and infrastructure monitoring. For example, the industrial parter that aided the evaluation of the results in this paper is currently facing the issue of continuously changing and re-arranging their big data stream processing application to several parameters, for example: (a) types of content that need analysis (multimedia images, audios as opposed to news articles and text); (b) types of users that need recommendation (e.g., governments as opposed to single users). Changing and constantly re-arranging an application's architecture requires constant and *continuous architecting* of architecture elements, their interconnection and their visible properties. Also, providing automated support to this continuous architecting exercise, reduces the (re-)design efforts and increases the speed of big data architectures' (re-)deployability by saving the effort of running trial-and-error experiments on expensive infrastructure.

This paper's contribution in support of said continuous architecting is twofold: (a) we elaborate a series of design anti-patterns and algorithmic manipulation techniques that would help designers identify problems in their designs; (b) we outline OSTIA, that stands for: "On-the-fly Static Topology Inference Analysis" - OSTIA allows designers and developers to infer the application architecture through on-the-fly reverse-engineering and architecture recovery [9]. During this inference step, OSTIA analyses the architecture to verify whether it is consistent with development restrictions and/or deployment constraints of the underlying development frameworks (e.g., DAG constraints). To do so, OSTIA hardcodes intimate knowledge on the streaming development framework (Storm, in our case) and its dependence structure in the form of a meta-model [10]. This knowledge is necessary to make sure

that elicited topologies are correct, so that models may be used in at least five scenarios: (a) realising an exportable visual representation of the developed topologies; (b) verifying structural constraints on topologies that would only become evident during infrastructure setup or runtime operation; (c) verifying the topologies against anti-patterns [11] that may lower performance and limit deployability/executability; (d) manipulate said topologies to elicit non-obvious structural properties such as linearisation or cascading; (e) finally, use topologies for further analysis, e.g., through model verification [12]. In an effort to offer said support in a DevOps fashion, OSTIA was engineered to act as an architecture recovery mechanism that closes the feedback loop between operational data architectures (Ops phase) and their refactoring phase (Dev phase). Currently, OSTIA focuses on Apache Storm, i.e., one of the most famous and established real-time stream processing engines [4], [13]. The core element of Storm, is called *topology*, which represents the architecture of the processing components of the application (from now we use topology and architecture interchangeably).

This paper outlines OSTIA, elaborating major usage scenarios, benefits and limitations. Also, we evaluate OSTIA using case-study research to conclude that OSTIA does in fact provide valuable insights for continuous architecting of streaming-based big data architectures.

II. Research Solution

This section outlines OSTIA starting form a brief recap of the technology it is currently designed to support, i.e., the Apache Storm framework. Further on, the section introduces how OSTIA was designed to support continuous architecting of streaming topologies focusing on Storm. Finally, the section outlines the meta-model for Storm that captures all restrictions and rules (e.g., for configuration, topology, dependence, messaging, etc.) in the framework. OSTIA uses this meta-model as a reference every time the application is run to recover and analyse operational topologies.

A. OSTIA design

The overall architecture of OSTIA is depicted in Figure 1. The logical architectural information of the topology is retrieved by OSTIA via static analysis of the source code. OSTIA generates a simple intermediate format to be used by other algorithmic processes.

OSTIA is architected in a way that algorithmic analysis, such as anti-pattern analyses, can be easily added. These analyses use the information resides in the intermediate format and provide added value analyses for continuous architecting of storm topologies. Since the information in the intermediate format only rely on the logical code analysis, the algorithmic analyses require some information regarding the running topology, such as end to end latency and throughput.

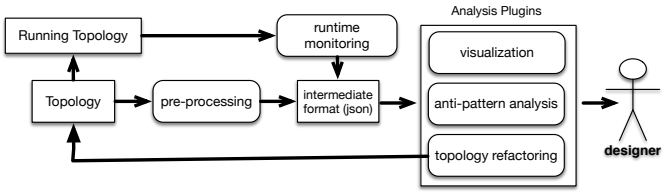


Fig. 1. OSTIA extensible architecture.

Such information will be continuously added to the intermediate repository via runtime monitoring of the topology on real deployment cluster. These provide appropriate and rich information for refactoring the initial architecture and enabling performance driven DevOps [14].

B. Storm Architecture

Storm is a technology developed at Twitter [13] in order to face the problem of processing of streaming of data. It is defined as a distributed processing framework which is able to analyse streams of data. The core element in the system is called *topology*. A Storm topology is a computational graph composed by nodes of two types: spouts and bolts. The former type includes nodes that process the data entering the topology, for instance querying APIs or retrieve information from a message broker, such as Apache Kafka. The latter executes operations on data, such as filtering or serialising.

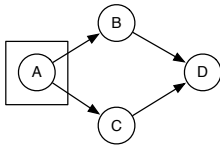
III. OSTIA-Based Continuous Architecting

This section elaborates on the ways in which OSTIA supports continuous architecting. First, we elaborate on the anti-patterns supported in OSTIA. Second, we elaborate on the algorithmic manipulation that OSTIA could apply to topologies to provide alternative visualisation. Third, finally, we elaborate on how OSTIA can drive continuous improvement assisted by formal verification. All figures in these sections use a simple graph-like notation where nodes may be any topological element (e.g., Spouts or Bolts in Apache Storm terms) while edges are directed data-flow connections.

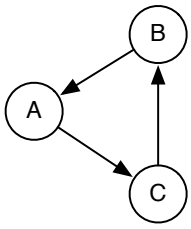
A. Topology Design Anti-Patterns Within OSTIA

This section elaborates on the anti-patterns we elicited through self-ethnography. These anti-patterns are elaborated further within OSTIA to allow for their detection during streaming topology inference analysis. Every pattern is elaborated using a simple graph-like notation where *spouts* are nodes that have outgoing edges only whereas *bolts* are nodes that can have either incoming or outgoing edges respectively.

1) Multi-Anchoring: The Multi-Anchoring pattern is shown in Fig. III-A1. In order to guarantee fault-tolerant stream processing, tuples processed by bolts need to be anchored with the unique id of the bolt and be passed to multiple acknowledgers (or "ackers" in short) in the topology. In this way, ackers can keep track of tuples in the topology. Multiple ackers can indeed cause much overhead and influence the operational performance of the entire topology.



(a) Multi-anchoring.



(b) Cycle-in.

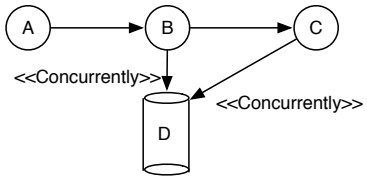


Fig. 2. Concurrency management in case of Persistent Data circumstances.

2) Cycle-in Topology: The Cycle-in pattern is shown in Fig. III-A1. Technically, it is possible to have cycle in Storm topologies. An infinite cycle of processing would create an infinite tuple tree and make it impossible for Storm to ever acknowledge spout emitted tuples. Therefore, cycles should be avoided or resulting tuple trees should be investigated additionally to make sure they terminate at some point and under a specified series of conditions. The anti-pattern itself may lead to infrastructure overloading and increased costs.

3) Persistent Data: The persistent data pattern is shown in Fig. 2. This pattern covers the circumstance wherefore if two processing elements need to update a same entity in a storage, there should be a consistency mechanism in place. OSTIA offers limited support to this feature, which we plan to look into more carefully for future work. More details on this support are discussed in the approach limitations section.

4) Computation Funnel: The computational funnel is shown in Fig. 3. A computational funnel emerges when there is not a path from data source (spout) to the bolts that sends out the tuples off the topology to another topology through a messaging framework or through a storage. This circumstance should be dealt with since it may compromise the availability of results under the desired performance restrictions.

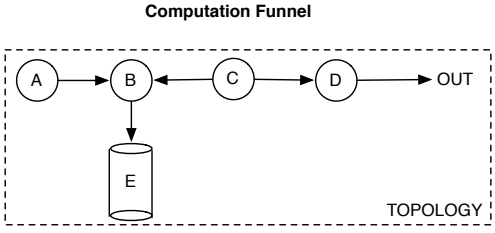


Fig. 3. computation funnel.

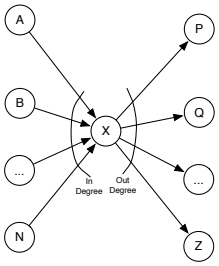


Fig. 4. Fan-in fan-out in Stream topologies.

B. Algorithmic Analysis on Stream Topologies

This section elaborates on the algorithmic analysis supported by OSTIA using the common graph-like notation introduced previously. OSTIA currently supports two topology content analysis (see Sec. III-B1 and III-B2) as well as two topology layout analyses (see Sec. III-B3 and III-B4). Only a part of these analyses is currently implemented in OSTIA. We discuss approach limitations further in Sect. V.

1) Fan-in/Fan-out: The Fan-in/Fan-out algorithmic manipulation is outlined in Fig. 4. For each element of the topology, fan-in is the number of incoming streams. Conversely, fan-out is the number outgoing streams. In the case of bolts, both in and out streams are internal to the topology. For Spouts, incoming streams are the data sources of the topology (e.g., message brokers, APIs, etc).

This algorithmic manipulation allows to visualise instances where fan-in and fan-out numbers are differing.

2) Topology cascading: The Cascading algorithmic manipulation is outlined in Fig. 5. By topology cascading, we mean connecting two different Storm topologies via a messaging framework (e.g., Apache Kafka [15]). Although cascading may simplify the development of topologies by encouraging architecture elements' reuse especially for complex but procedural topologies, this circumstance may well raise the complexity of continuous architecting and may require separation of concerns [16]. For example, Fig. 5 outlines an instance in which two topologies are concatenated together by a message broker. In this instance, formal verification may be applied on the left-hand side topology, which is more business-critical, while the right-hand side of the entire topology is improved by on-the-fly OSTIA-based analysis. Even though OSTIA support for this feature is still limited, we report it nonetheless since OSTIA was engineered to address multiple topologies at once.

This algorithmic manipulation allows to combine multiple cascading topologies.

3) Topology clustering: Topology clustering is outlined in Fig. 6. Topology clustering implies identifying coupled processing elements (i.e., bolts and spouts) and cluster them together (e.g., by means of graph-based analysis) in a way that elements in a cluster have high cohesion and loose-coupling with elements in other clusters. Simple clustering or Social-Network Analysis mechanisms can be used to infer clusters. These clusters may require additional attention since they

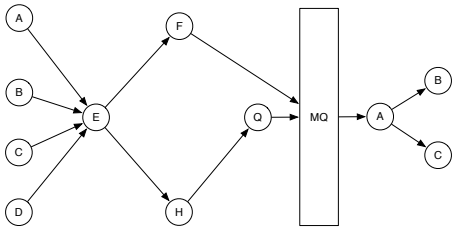


Fig. 5. cascading.

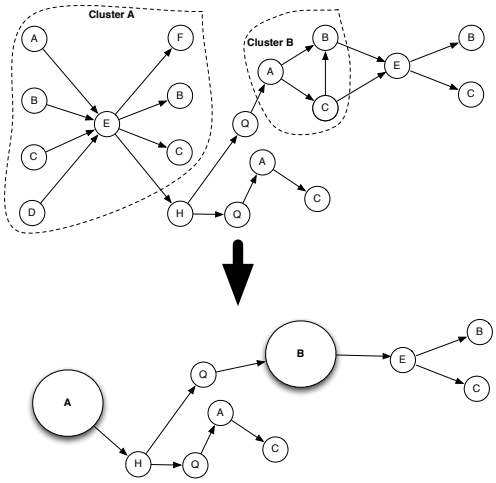


Fig. 6. clustering.

could turn out to become bottlenecks. Reasoning more deeply on clusters and their resolution may lead to establishing the Storm scheduling policy best-fitting with the application.

4) Linearising a topology: Topology linearisation is outlined in Fig. 7. Sorting the processing elements in a topology in a way that topology looks more linear, visually. This step ensures that visual investigation and evaluation of the structural complexity of the topology is possible by direct observation. It is sometimes essential to provide such a visualisation to evaluate how to refactor the topology as needed.

IV. Evaluation

We evaluated OSTIA through qualitative evaluation and case-study research featuring an open-/closed-source industrial case study (see Section IV-A) and two open-source case studies (see Section IV-B).

A. Industrial Case-Study

OSTIA was evaluated using several topologies part of the SocialSensor App. Our industrial partner is having performance and availability outages connected to currently unknown circumstances. Therefore, the objective of our evaluation for OSTIA was twofold: (a) allow our industrial partner to enact continuous architecting of their application with the goal of discovering any patterns or hotspots that may be requiring further architectural reasoning; (b) understand

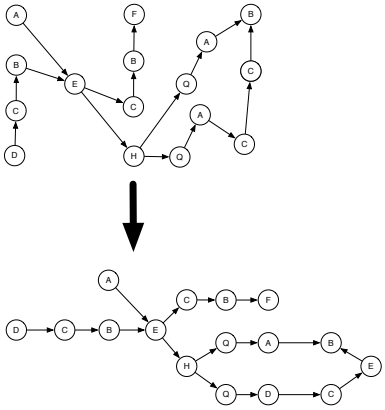


Fig. 7. linearising.

whether OSTIA provided valuable feedback to endure the continuous architecting exercise.

OSTIA standard output[1] for the smallest of the three SocialSensor topologies, namely the "focused-crawler" topology, is outlined in Fig. 8.

OSTIA has been proved particularly helpful in visualising the complex topology together with the parallelism level of each components. Combining this information with runtime data, such as latency, our industrial partner observed that the "expander" bolt needed additional architectural reasoning. Also, the partner welcomed the idea of using OSTIA as a mechanism to enact continuous architecting of the topology in question as part of the needed architectural reasoning.

Besides this pattern-based evaluation and assessment, OSTIA algorithmic analyses assisted our client in understanding that the topological structure of the SocialSensor app would be better fit for batch processing rather than streaming, since the partner observed autonomously that too many database-output spouts and bolts were used in their versions of the SocialSensor topologies. In so doing, the partner is now using OSTIA to drive the refactoring exercise towards a Hadoop Map Reduce [6] framework for batch processing.

B. Evaluation on Open-Source Software

To confirm the usefulness and capacity of OSTIA to enact a continuous architecting cycle, we applied it in understanding (first) and attempting improvements of two open-source applications, namely, the previously introduced DigitalPebble [17] and StormCV [18] applications. Figures 9 and 10 outline standard OSTIA output for the two applications. Note that we did not have any prior knowledge concerning the two applications in question and we merely run OSTIA on the applications' codebase dump in our own experimental machine. OSTIA output takes mere seconds for small to medium-sized topologies (e.g., around 25 nodes).

The OSTIA output aided as follows: (a) the output summarised in Fig. 9 allowed us to immediately grasp the func-

[1]Output of OSTIA analyses is not evidenced for the sake of space.

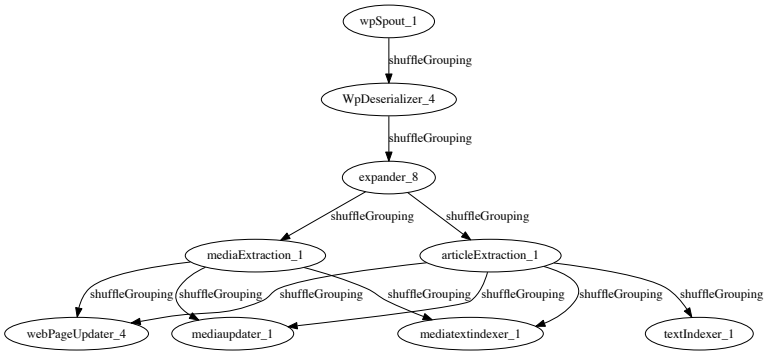


Fig. 8. SocialSensor App, OSTIA sample output.

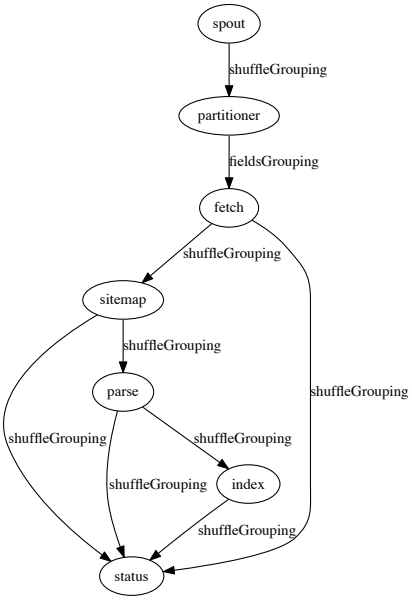


Fig. 9. DigitalPebble topology.

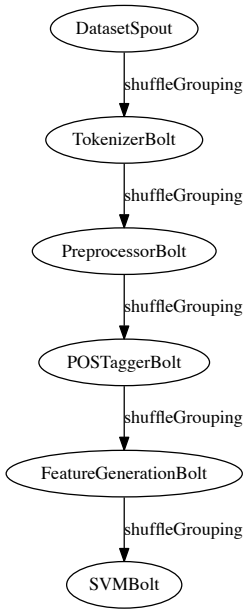


Fig. 10. StormCV topology.

tional behavior of the DigitalPebble and StormCV topologies allowing us to interpret correctly their operations before reading long documentation or inspecting the code; (b) OSTIA aided us in visually interpreting the complexity of the applications at hand; (c) OSTIA allowed us to spot several anti-patterns in the DigitalPebble Storm application around the "sitemap" and "parse" bolts, namely, a multiple cascading instance of the multi-anchoring pattern and a persistent-data pattern. Finally, OSTIA aided in the identification of the computational funnel anti-pattern around the "status" bolt closing the DigitalPebble topology. With this evaluation at hand, developers in the respective communities of DigitalPebble and StormCV could refactor their topologies, e.g., aided by OSTIA-based formal verification that proves the negative effects of said anti-patterns.

V. Discussion

This section discusses some findings and the limitations of OSTIA.

OSTIA represents one humble, but significant step at supporting practically the necessities behind developing and maintaining high-quality big-data application architectures. In designing and developing OSTIA we encountered a number of insights that may aid continuous architecting.

First, we found (and observed in industrial practice) that it is often more useful to develop a quick-and-dirty but

"runnable" architecture topology than improving the topology at design time for a tentatively perfect execution. This is mostly the case with big-data applications that are developed stemming from previously existing topologies or applications. OSTIA hardcodes this way of thinking by supporting reverse-engineering and recovery of deployed topologies for their incremental improvement. Although we did not carry out extensive qualitative or quantitative evaluation of OSTIA in this regard, we are planning additional industrial experiments for future work with the goal of increasing OSTIA usability and practical quality.

Second, big-data applications design is an extremely young and emerging field for which not many software design patterns have been discovered yet. The (anti-)patterns and approaches currently hardcoded into OSTIA are inherited from related fields, e.g., pattern- and cluster-based graph analysis. Nevertheless, OSTIA may also be used to investigate the existence of recurrent and effective design solutions (i.e., design patterns) for the benefit of big-data application design. We are improving OSTIA in this regard by experimenting on two fronts: (a) re-design and extend the facilities with which OSTIA supports anti-pattern detection; (b) run OSTIA on multiple big-data applications stemming from multiple technologies beyond Storm (e.g., Spark, Hadoop Map Reduce, etc.) with the purpose of finding recurrent patterns. A similar approach may feature OSTIA as part of architecture trade-off analysis campaigns [19].

Third, a step which is currently undersupported during big-data applications design is devising an efficient algorithmic breakdown of a workflow into an efficient topology. Conversely, OSTIA does support the linearisation and combination of multiple topologies, e.g., into a cascade. Cascading and similar super-structures may be an interesting investigation venue since they may reveal more efficient styles for big-data architectures beyond styles such as Lambda Architecture [20]. OSTIA may aid in this investigation by allowing the interactive and incremental improvement of multiple (combinations of) topologies together.

VI. Conclusion

We set out to assist the continuous architecting of big data streaming designs by OSTIA, a toolkit to assist designers and developers to facilitate static analysis of the architecture and provide automated constraint verification in order to identify design anti-patterns and provide structural refactorings. OSTIA helps designers and developers by recovering and analysing the architectural topology on-the-fly, assisting them in: (a) reasoning on the topological structure and how to refine it; (b) export the topological structure consistently with restrictions of their reference development framework so that further analysis (e.g., formal verification) may ensue. In addition, while performing on-the-fly architecture recovery, the analyses that OSTIA is able to apply focus on checking for the compliance to essential consistency rules specific to targeted big data frameworks. Finally, OSTIA allows to check whether the recovered topologies contain occurrences of key anti-patterns. By running a case-study with a partner organization, we observed that OSTIA assists designers and developers in establishing and continuously improving the quality of topologies behind their big data applications. We confirmed this result running OSTIA on several open-source applications featuring streaming technologies. We released OSTIA as an open-source software [21].

In the future we plan to further elaborate the anti-patterns, exploiting graphs analysis techniques inherited from social-networks analysis. Also, we plan to expand OSTIA to support further technologies beyond the most common application framework for streaming, i.e., Storm. Finally, we plan to further evaluate OSTIA using empirical evaluation.

Acknowledgment: The Authors' work is partially supported by the European Commission grant no. 644869 (EU H2020), DICE. Also, Damian's work is partially supported by the European Commission grant no. 610531 (FP7 ICT Call 10), SeaClouds.

References

[1] C. K. Emani, N. Cullot, and C. Nicolle, "Understandable big data: A survey." *Computer Science Review*, vol. 17, pp. 70–81, 2015.
[2] B. Ratner, *Statistical and Machine-Learning Data Mining: Techniques for Better Predictive Modeling and Analysis of Big Data*. CRC PressINC, 2012.
[3] [Online]. Available: http://www.gartner.com/newsroom/id/2637615
[4] R. Evans, "Apache storm, a hands on tutorial." in *IC2E*. IEEE, 2015, p. 2.
[5] [Online]. Available: http://spark.apache.org/
[6] [Online]. Available: https://hadoop.apache.org/
[7] D. A. Tamburri, P. Lago, and H. van Vliet, "Organizational social structures for software engineering." *ACM Comput. Surv.*, vol. 46, no. 1, p. 3, 2013.
[8] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A Software Architect's Perspective.*, ser. SEI series in software engineering. Addison-Wesley, 2015.
[9] I. Ivkovic and M. W. Godfrey, "Enhancing domain-specific software architecture recovery." in *IWPC*. IEEE Computer Society, 2003, pp. 266–273.
[10] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002.
[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture. Vol. 2, Patterns for concurrent and networked objects*. John Wiley, Chichester, England, 2000.
[12] M. M. Bersani, S. Distefano, L. Ferrucci, and M. Mazzara, "A timed semantics of workflows." in *ICSOFT (Selected Papers)*, ser. Communications in Computer and Information Science, vol. 555. Springer, 2014, pp. 365–383.
[13] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.
[14] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski *et al.*, "Performance-oriented devops: A research agenda," *arXiv preprint arXiv:1508.04752*, 2015.
[15] [Online]. Available: http://kafka.apache.org/
[16] T. Mens and M. Wermelinger, "Issue overview: Separation of concerns in software evolution," vol. 14, no. 5, pp. 311–315, 2002.
[17] [Online]. Available: https://github.com/DigitalPebble
[18] [Online]. Available: https://github.com/sensorstorm/StormCV
[19] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001.
[20] M. Quartulli, J. Lozano, and I. G. Olaizola, "Beyond the lambda architecture: Effective scheduling for large scale eo information mining and interactive thematic mapping." in *IGARSS*, 2015, pp. 1492–1495.
[21] [Online]. Available: https://github.com/maelstromdat/OSTIA