Control plane systems tracing and debugging – existing problems and proposed solution

Gleb Peregud[1], Maria Ganzha[1][0000−0001−7714−4844], and Marcin Paprzycki[2][0000−0002−8069−2152]

[1] Warsaw University of Technology, Warsaw, Poland
M.Ganzha@mini.pw.edu.pl
[2] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
marcin.paprzycki@ibspan.waw.pl

Abstract. Hierarchical control plane systems are hard to debug and reason about, among others, because of prevalence of intent-driven actuation. Moreover, an industry adopted distributed systems tracing model, called OpenTracing, does not handle activity tracing in presence of coalescing effects, materializing in control plane systems, e.g. cloud platforms and build systems. The goal of this contribution is to outline a solution for reasoning about such systems, by creating a novel distributed systems tracing mechanism, based on an extension of the OpenTracing model.

Keywords: Distributed systems tracking · Provenance tracking · Hierarchical control plane systems · Build systems.

1 Introduction

Complexity of computer systems is growing [11], and results in individuals' losing ability to fully comprehend systems they create/use. Today, programs may be developed by tens of thousands developers [31,12]. Moreover, there exist services with millions of lines of code, with hundreds of Remote Procedure Call interfaces [1]. Growth of system complexity is followed by increasing complexity of software automation. Today, it ranges from simple build systems (Make [13]), through imperative automation systems (Puppet [22]), to complex multi-cloud systems (Terraform [5]), which become too complex for individual developers. For instance, understanding a complex service deployment would require expertise in Terraform, Docker [25], Linux kernel, Kubernetes [7], cloud provider API(s), and deployment requirements of the servers. Among the most complex automation systems, are the cloud services [23,44]. They serve public APIs to cloud customers and drive the cloud infrastructure.

1.1 Intent-driven actuation

Deployment systems are shifting from an imperative execution model (like Ansible), to a declarative-first model (like Terraform). They follow a scheduling policy to reach an intended state, executing a sequence of, usually small/restricted,

imperative steps. We call these *intent-based systems*, and their execution an *intent-driven actuation*. For example, Kuberentes is an intent-based system: its API objects [21] describe the desired state (user intent) and the system match updates the cluster to match it. Furthermore, deployment systems (NixOps [10]) and modern build systems ([27,24]) employ intent-driven actuation. They build a new instance of a target object from the scratch, based on the specification, and replace the current object with the new one (imperative operation).

1.2 Coalescing effects

Systems which employ intent-driven actuation exhibit the *coalescing effects*. In general, coalescing occurs when work units (from multiple requests), are batched, before joint execution [34]. In build systems, coalescing effect occurs in a "diamond dependency graph", since the shared dependency is built in response to both incoming requests. Here, no single attribution of causality is possible. Overall, coalescing effects change relationship, between requests and activities is the system, from one-to-many to many-to-many.

In intent-driven actuation, coalescing effects appear as actuation aggregates "multiple intents". Here, actuation can take an "arbitrary path" between the current state and the desired state. The system can choose to batch user requests and satisfy them all at once, to execute multiple actions based on a single request, or do "something in-between". Let us illustrate this with a few examples:

- If two consecutive requests desire the same state, only one is acted upon.
- If two consecutive requests modify the same property, in a quick succession, only the latter actuation is likely to occur, due to the, so called, debouncing.
- If three requests modify the state in quick succession from A to B, to C, and back to B, typically only actuation from A to B will be executed.
- If a request considerably changes a desired state, automation system will perform a series of actions over long period of time. For example to deliver 10,000 VMs the cloud will not start them instantaneously, but over time.

Observe that a combination of such behaviours results in *arbitrary relationships between requests and actuations*. This makes majority of debugging tools inaccurate. In this context, a widely adopted industry approach to reasoning is *distributed systems tracing*, and the most popular tools follow OpenTracing model [29] based on Google's Dapper. However, this model does not handle coalescing effects (see [34]).

1.3 Control plane systems

Coalescing effects are present in *control plane systems*, in upper layers of cloud service stacks. Cloud APIs can be imperative, declarative, or both. Imperative API allows direct actions on cloud resources. Declarative API allows users to declare an arbitrarily complex intent with a single call. An example of such desired intent is a shape of a service deployment, e.g. size and regional distribution of a

group of virtual machines. Typically, declarative API calls trigger asynchronous work changing the deployment to match the desired state.

Additionally, in Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) solutions, deployment systems may be provided as services. For example, any PaaS offering Kubernetes API, exposes an intent-based solution. It can be assumed that PaaS offerings from Amazon Web Services, Google, or Azure combine their systems scale with complex execution model of intent-based systems.

1.4 Hierarchical control plane systems

In deployment practice, some cloud service APIs are implemented on top of other cloud services. For example, Google Cloud Functions (GCF) runs on top of Google Cloud Run (GCR), which runs on top of Google Kuberentes Engine (GKE). In turn, GKE runs on top of Google Cloud Engine (GCE). Hence control plane system may be *hierarchical*, and the hierarchy can have quite a complex structure of dependencies making it hard to be reasoned about.

1.5 Build systems in control plane systems

Let us now consider that control plane systems, with declarative APIs, need to be provided with the desired state. Build system outputs are often used for this purpose. For example, Infrastructure-as-Code (IasC) model recommends treating service deployment as any other code – to be be built and tested using *build systems*. Here, Bazel build system can define desired shape of Kubernetes deployments [3]Immutable infrastructure [26] model often prescribes use of Docker containers and Docker's primitive build system [27].

Furthermore, *build/build-like systems* are often parts of control plane systems. For example, Terraform Cloud [16] and EPAM Cloud [37] execute Terraform planning process as part of their API. Tools like NixOps [10] and Disnix [41]define deployments end-to-end, starting with individual binaries, through the VM/container images, to the "shape of the cloud".

2 Problem statement

Hierarchical control plane systems employing intent-driven actuation (HCPS) are hard to reason about. These are multi-layered systems, with a large number of servers. Hence, reasoning about them requires tracing both their activities and mutations of the state, managed by any layer of the system. A control plane system in a cloud service is inevitably multi-tenant. Behaviour of individual tenants can affect how other tenants are being served, and can affect the health of the system. Since such interactions are difficult to reproduce, tools providing information about the control plane system must be always on. Understanding system behavior requires observing related activities and associated state changes across many services and machines. Given that these systems are developed by large organizations with multiple teams, it may not be obvious which services are in use.

The aim of undertaken work is to propose a debugging solution, applicable to hierarchical control plane system employing intent-driven actuation.

3 Related work

To address this problem, it is necessary to collect data about the state of a HCPS. Here, two main approaches can be identified: *distributed systems tracing*, *monitoring and logging*, *provenance tracking*, and *build systems*.

3.1 Distributed systems tracing

The state-of-the-art solutions for *distributed tracing* closely follow the OpenTracing model. The OpenTracing approach uses "tainting", where an incoming request is "tainted" with a unique identifier. It is propagated across all activities participating in handling the request. This has been standardized by W3C as the Trace Context recommendation [20].

Tainting propagates a single identifier only, hence it cannot deal with coalescing effects. None of OpenTainting open source implementations solve to this problem. Let us note that intent-driven actuation cannot be faithfully recorded using tree-shaped execution traces, where an actuation may be linked with more than one incoming request. Hence intent-based systems **necessitate a more flexible model**. Note also that the tainting model cannot be extended by propagating of a set of trace identifiers instead of a single one, due to super-linear storage requirements and duplication of spans.

3.2 Monitoring and logging

Monitoring and logging mechanisms allow capturing metrics and unstructured log entries. Metrics are most suitable to capture aggregated statistics of the system. They can be labeled with additional information, allowing finer-grained view, but they still deliver aggregated numbers [38]. Logging mechanisms are typically based on unstructured log entries. Neither mechanism is suitable to track relationships between incoming requests and individual work units.

3.3 Provenance tracking

Coalescing effects are associated with shared mutable states. Hence, provenance tracking becomes relevant, as it allows tracking state objects, their mutations and relations. It is related to database provenance (data lineage tracking). Provenance traces [8] track provenance at query time. This is a bottom-up approachusing the proposed formal language. The solution generates very large traces, as all transformations are recorded. If applied naively, this approach is not feasible for large-scale control plane system, since it (1) requires writing from scratch, and (2) does not allow precise specification, which tracking data is to be gathered.

It has been recognized [19] that provenance tracking in clouds requires aggregation across multiple layers of the environment. Provenance tracking has been applied in intent-based control plane systems in networking. Intent-based networking (IBN) concerns automated and policy-aware network management [35]. ProvIntent [39] is a framework extension for the SDN control plane that accounts for intent semantics. It extends the ProvSDN [40], to explicitly incorporate intent evolution in provenance tracking. It uses W3C PROV data model.

W3C PROV expands on OpenProvenance model (OPM) [28] as data model for provenance on the Web. It is sufficiently expressive to represent coalescing effects – it can represent non-tree-shaped activities and it can record objects as triggers for activities. However, the PROV model is not suitable since: (1) Most activities in HCPSs do have a tree-shaped control flow. Here, PROV Activities have to be represented using *wasStartedBy* and *wasEndedBy* relations, causing substantial storage overhead. (2) While [15] proposes use of OPM to represent message passing in distributed systems, proposed representation is verbose compared to OpenTracing. (3) PROV model is expressive and complex much beyond OpenTracing. This complexity could hinder its adoption in industry. Systems like SPADE [14] do use PROV model to capture provenance data in distributed settings. SPADE is focused on gathering low-level information from OS audit logs, network artifacts, etc. Therefore, SPADE's syscall and library call level instrumentation would not scale for a large production system.

Provenance tracking has been researched from the perspective of *formal systems*. Here, Souilah et.al. [36] presented a formal provenance, based on the π-calculus. The approach is based on enriching exchanged data with provenance information, similar to the tainting approach. Hence, this approach is not suitable for our problem. Why-across-time provenance ([43]) provides state machines based mechanisms to track data provenance in time-varying stateful distributed systems. However, wat-provenance requires determinism, while large scale distributed systems involve a fair share of non-determinism in load balancing, bin-packing, resource allocation, load shedding, etc. Additionally both this and π -calculus-based formalism depart from the well-established OpenTracing.

Provenance tracking in security is focused on threat detection. This puts constraints on available approaches. ProTracer [33] uses a mix of logging and tainting, and is focused on Advanced Threat Protection. Hence, it uses kernel-level audit logging and syscall interception, as a black-box, zero-trust approach. ProTracer does not use a general data model, making it impossible to represent abstract entities, e.g. cloud resources. CamFlow [30] automates provenance capture, as a Linux Security Module (LSM), designed for single-machine system auditing. It outputs results in W3C PROV format.

Security-focused provenance tracking addresses different issues than these in HCPSs. Security requires zero trust, while the HCPSs case allows for full trust. It restricts security provenance solutions to use *observed* provenance. However, provenance tracking in large-scale control plane systems cannot be based solely on observed provenance, due to use of abstract entities and their scale [32]. Provenance tracking in HCPSs should rather apply *disclosed* provenance.

Provenance has been approach in the context of workflow management systems and data transformation pipelines. RAMP is a data intensive scalable computing (DISC) provenance framework [18]. It is restricted to data intensive computations, over static data. Therefore, it cannot be applied to components, like storage systems, coordination services, load balancers, etc., where control flow evolves with the system, and data passes via a large set of mechanisms (RPCs, databases, pubsub systems, etc). LogProv is a provenance logging system, implemented for Apache Pig and Apache Hadoop [42]. It supports dynamically shaped big data workflows and pipelines. It uses structured logging and the ElasticSearch. Overall, RAMP is a very specialized framework for DISC provenance tracking, while LogProv is less specialized and more flexible. Here, we believe that LogProv approach can be generalized.

3.4 Build systems

As discussed, build systems play important role in HCPSs, while reasoning about build systems combines tracing and provenance tracking. Build systems are inherently intent-based, since a build target is described declaratively. Moreover, memorization and incremental recomputation are coalescing effects, used to achieve a minimality property [27]. Here, build systems, like Bazel [2] or Nix [9], provide mechanisms to reason about, debug and optimize operations. These mechanisms are implementation-specific, informal, and not introperable.

Separately, tracing build processes has been used to uncover licence compliance inconsistencies [6], which is a form of provenance tracing. This approach does not address hierarchical nature of control plane systems, and captures only two specialized levels of provenance for build tasks and files used in builds.

Finally, build system dependency tracking can be seen as provenance tracking, for build targets based on their inputs (disclosed provenance in [4]). However, at best, it works at the level of the build graph, which is too high level of granularity. For instance, when build systems operate within a control plane system, ability to track relationship between build system inputs and actions of a control plane components taken based on build system outputs is necessary.

4 Solution outline

We will now outline the requirements towards a system that can solve the posed problem. Here, we believe that the solution should extend OpenTracing (to facilitate adoption). Hence, such solution lies between the OpenTracing and the OpenProvenance in the design space of debuggability tools. Hence, based on analysis of literature and existing industrial tools, the proposed solution should have the following properties:

1. *Coalescing effects support* – to support tracing of intent-driven actuation;
2. *Support for abstract entities* – essential to align with cloud APIs;
3. *Support for composite entities* – to support objects like archives, VM images, container images, etc. which are prevalent in cloud APIs;

4. *Low storage overhead* – necessary for large-scale systems;
5. *Full coverage* – for all tracked activities and resource state mutations;
6. *Gradual fidelity execution tracing* – to selectively apply execution tracing;
7. *Gradual fidelity provenance tracking* – to selectively track provenance;
8. *Minimal mental burden* – to support adoption in industry;
9. *Cross-host tracking* – for distributed systems tracing support;
10. *Multi-layer systems support* – to allow tracking across hierarchical layers;
11. *Asynchronous data intake* – to support data ingestion in presence of unreliable network, unpredictable latencies, and lack of ordering guarantees;
12. *Event-based data production* – to deal with faults and to avoid buffering;
13. *Flexible control flow support* – to deal with pre-existing control flows.

We believe that a system, which satisfies these requirements, will be able to trace (1) non-ultra-large on-line serving systems, and (2) hierarchical control plane systems employing intent-driven actuation. Note that existing solutions, like Dapper or CamFlow, satisfy only a subset of these requirements.

4.1 Provenance-enhanced distributed systems tracing model

We propose Provenance-Enhanced Distributed Systems Tracing (PEDST) model, as a foundation for the solution. PEDST extends the OpenTracing model to record interactions between actions in the system and objects the system interacts with. Hence, the following concepts are used in PEDST model. *Execution* (similar to the OpenTracing "span"), tracks *operations* on *entities*, which are recorded. *Entity* is "anything" that is "important-enough" to track. Read and write *operations*, performed by an *execution*, allow tracking provenance of objects they operate on. Each write *operation*, on an entity, gives rise to a new *incarnation* (immutable *entity* revision). *Entities* are mutable. *Executions* of the same logic are grouped, by association with a *process*, which describes a procedure. A pair of *executions* can interact. Recorded *interaction* consists of *messages*. A *message* may carry an *incarnation* as a payload.

The PEDST model supports both *activity tracing* and *provenance tracking*. The vehicles of tracing are *executions* and their parent-child relationships [34]. Correspondingly, the vehicle for provenance tracking are read and write operations performed by *incarnations*. This puts the proposed approach in the "data provenance" class [17]. Capability to dynamically track provenance of objects (e.g. files, configs, resources, etc.) contributes to "support for abstract entities".

Concepts of *executions*, *incarnations*, and *operations*, are sufficient to perform both execution tracing and provenance tracking. Concepts of *process* and *entity* allow improving the usability of envisioned tools. *Annotations* are necessary for the model to be a superset of OpenTracing. *Interactions* and *messages* are necessary to track provenance propagation through RPCs and other message passing mechanisms found in distributed systems.

Furthermore, use of OpenTracing as the foundation, allows to partially satisfy the "minimal mental burden" requirement, due to familiarity with this model. Nevertheless PEDST is more complex than OpenTracing. However, this complexity is inherent to the problem. Here, the *entities* concept, and versioning

with *incarnations*, and *operations* as a link between activity tracing and provenance tracking, are the minimal extension of the OpenTracing model, to make it comply with other requirements.

4.2 Architecture

Finally, to solve the problem the model needs to be implemented as part of a software system. We propose to use an architecture akin go Dapper's or LogProv's, but extended accordingly. The data model described above is used to represent the final processed data, while data collection from individual servers is done with a distributed structured logging mechanism, using a logging data model. The logging data model needs to support aggregation into the PEDST model, by a log-tailing processors. Processor ingests data from the logging mechanism and stores it into the PEDST storage. Optionally, data can be transformed into a graph-based representation, for a more flexible querying support. Additionally a visualization tool needs to expose the processed data to a user of the system.

5 Conclusions

In this work we have identified characteristics of hierarchical control plane systems, with intent-driven actuation, which makes their debuggability an unsolved problem. Next, we have identified *provenance* as a possible solution. We have introduced main aspects of a provenance-enhanced distributed systems tracing model; an extension of the Dapper tracing model, with elements of provenance tracking. Finally an architecture of a software implementation was suggested. Further work on research, implementation and application of the proposed model is under way and will be reported in subsequent publications.

References

1. Barroso, L.A., Ranganathan, P.: Datacenter-scale computing. IEEE Micro **30**, 6–7 (2010), http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.63, special issue of the IEEE Micro Magazine
2. Optimizing performance (May 2020), https://docs.bazel.build/versions/master/skylark/performance.html, [Online; accessed 30. May 2020]
3. This repository contains rules for interacting with kubernetes configurations / clusters. (2020), https://github.com/bazelbuild/rules_k8s, [Online; accessed 30. May 2020]
4. Braun, U., Garfinkel, S., Holland, D.A., Muniswamy-Reddy, K.K., Seltzer, M.I.: Issues in automatic provenance collection. In: International Provenance and Annotation Workshop. pp. 171–183. Springer (2006)
5. Brikman, Y.: Terraform: Up & Running: Writing Infrastructure as Code. O'Reilly Media (2019)
6. van der Burg, S., Dolstra, E., McIntosh, S., Davies, J., German, D.M., Hemel, A.: Tracing software build processes to uncover license compliance inconsistencies. Proceedings of the 29th ACM/IEEE international conference on Automated software engineering pp. 731–742 (9 2014). https://doi.org/10.1145/2642937.2643013

7. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Queue **14**(1), 70–93 (2016)
8. Cheney, J., Acar, U., Ahmed, A.: Provenance traces. arXiv preprint arXiv:0812.0564 (2008)
9. Dolstra, E.: The purely functional software deployment model. Utrecht Uni. (2006)
10. Dolstra, E., Vermaas, R., Levy, S.: Charon: Declarative provisioning and deployment. In: 2013 1st International Workshop on Release Engineering (RELENG). pp. 17–20. IEEE (2013)
11. Dvorak, D.: Nasa study on flight software complexity. In: AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference. p. 1882 (2009)
12. Facebook Engineering: 9.9 million lines of code and still moving fast - facebook open source in 2014 (2014), https://engineering.fb.com/core-data/9-9-million-lines-of-code-and-still-moving-fast-facebook-open-source-in-2014/
13. Fowler, G.: A case for make. Software: Practice and Experience **20**(S1), S35–S46 (1990). https://doi.org/10.1002/spe.4380201305, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380201305
14. Gehani, A., Tariq, D.: Spade: support for provenance auditing in distributed environments. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. pp. 101–120. Springer (2012)
15. Groth, P., Moreau, L.: Representing distributed systems using the Open Provenance Model. Future Gener. Comput. Syst. **27**(6), 757–765 (Jun 2011). https://doi.org/10.1016/j.future.2010.10.001
16. HashiCorp: Terraform (Aug 2020), https://www.terraform.io/docs/cloud/index.html, [Online; accessed 1. Sep. 2020]
17. Herschel, M., Diestelkämper, R., Lahmar, H.B.: A survey on provenance: What for? what form? what from? The VLDB Journal **26**(6), 881–906 (2017)
18. Ikeda, R., Park, H., Widom, J.: Provenance for generalized map and reduce workflows. CIDR pp. 273–283 (01 2011)
19. Imran, M., Hlavacs, H., Khan, F.A., Jabeen, S., Khan, F.G., Shah, S., Alharbi, M.: Aggregated provenance and its implications in clouds. Future Generation Computer Systems **81**, 348–358 (2018)
20. Kanzhelev, S., McLean, M., Reitbauer, A., Drutu, B., Molnar, N., Shkuro, Y.: Trace Context (2 2020), https://www.w3.org/TR/trace-context, [Online; accessed 1. Sep. 2020]
21. Understanding kubernetes objects (2020), https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/, accessed on 2020-05-13
22. Loope, J.: Managing infrastructure with puppet: configuration management at scale. " O'Reilly Media, Inc." (2011)
23. Low, C., Chen, Y., Wu, M.: Understanding the determinants of cloud computing adoption. Industrial management & data systems (2011)
24. McNerney, P.J.: Beginning Bazel. Apress, Berkeley, CA (2020). https://doi.org/10.1007/978-1-4842-5194-2
25. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux journal **2014**(239), 2 (2014)
26. Mikkelsen, A., Grønli, T.M., Kazman, R.: Immutable infrastructure calls for immutable architecture. In: Proceedings of the 52nd Hawaii International Conference on System Sciences (2019)
27. Mokhov, A., Mitchell, N., Peyton Jones, S.: Build systems à la carte: Theory and practice. Journal of Functional Programming **30** (1 2020).

https://doi.org/10.1017/s0956796820000088, https://oadoi.org/10.1017/s0956796820000088
28. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., et al.: The open provenance model core specification (v1. 1). Future generation computer systems **27**(6), 743–756 (2011)
29. The OpenTracing project (1 2020), https://opentracing.io, [Online; accessed 7. Jun. 2020]
30. Pasquier, T., Han, X., Goldstein, M., Moyer, T., Eyers, D., Seltzer, M., Bacon, J.: Practical whole-system provenance capture. In: Symposium on Cloud Computing (SoCC'17). ACM (2017)
31. Potvin, R., Levenberg, J.: Why google stores billions of lines of code in a single repository. Commun. ACM **59**(7), 78–87 (Jun 2016). https://doi.org/10.1145/2854146, https://doi.org/10.1145/2854146
32. Raju, B., Elsethagen, T., Stephan, E., Van Dam, K.K.: A scientific data provenance api for distributed applications. In: 2016 International Conference on Collaboration Technologies and Systems (CTS). pp. 104–111. IEEE (2016)
33. Shiqing, M., Zhang, X., Xu, D.: Protracer: Towards practical provenance tracing by alternating between logging and tainting. In: NDSS (01 2016). https://doi.org/10.14722/ndss.2016.23350
34. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc. (2010), https://research.google.com/archive/papers/dapper-2010-1.pdf
35. Sivakumar, K., Chandramouli, M.: Concepts of network intent. Internet Research Task Force, Internet Draft, Oct (2017)
36. Souilah, I., Francalanza, A., Sassone, V.: A formal model of provenance in distributed systems. In: Workshop on the Theory and Practice of Provenance. pp. 1–11 (2009)
37. Terraform as a service (2020), https://cloud.epam.com/site/competency_center/e=p=c_services/terraform_as_a_service(=t=a=s), accessed on 2020-05-30
38. Turnbull, J.: Monitoring with Prometheus. Turnbull Press (2018)
39. Ujcich, B.E., Bates, A., Sanders, W.H.: Provenance for intent-based networking. In: Proceedings of the IEEE Conference on Network Softwarization (2020)
40. Ujcich, B.E., Jero, S., Edmundson, A., Wang, Q., Skowyra, R., Landry, J., Bates, A., Sanders, W.H., Nita-Rotaru, C., Okhravi, H.: Cross-app poisoning in software-defined networking. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 648–663 (2018)
41. Van Der Burg, S., Dolstra, E.: Disnix: A toolset for distributed deployment. Science of Computer Programming **79**, 52–69 (2014)
42. Wang, R., Sun, D., Li, G., Atif, M., Nepal, S.: Logprov: Logging events as provenance of big data analytics pipelines with trustworthiness. In: 2016 IEEE International Conference on Big Data (Big Data). pp. 1402–1411. IEEE (2016)
43. Whittaker, M., Teodoropol, C., Alvaro, P., Hellerstein, J.M.: Debugging distributed systems with why-across-time provenance. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 333–346 (2018)
44. Wood, K., Anderson, M.: Understanding the complexity surrounding multitenancy in cloud computing. In: 2011 IEEE 8th International Conference on e-Business Engineering. pp. 119–124. IEEE (2011)