Ovid: A Software-Defined Distributed Systems Framework

Deniz Altınbüken
Cornell University

Robbert van Renesse
Cornell University

Abstract

We present Ovid, a framework for building evolvable large-scale distributed systems that run in the cloud. Ovid constructs and deploys distributed systems as a collection of simple components, creating systems suited for containerization in the cloud. Ovid supports evolution of systems through *transformations*, which are automated refinements. Examples of transformations include replication, batching, sharding, and encryption. Ovid transformations guarantee that an evolving system still implements the same specification. Moreover, systems built with transformations can be combined with other systems to implement more complex infrastructure services. The result of this framework is a *software-defined distributed system*, in which a logically centralized controller specifies the components, their interactions, and their transformations.

1 Introduction

Containerization in the cloud is a lightweight virtualization technique that is getting popular and helping developers build and run applications comprising multiple systems in the cloud. The container technology creates an ecosystem where systems can be deployed quickly and can be composed with other systems. While we have a good understanding of how to build cloud services that are comprised of many systems, we do not have the technology to reason about how these systems can be reconfigured and evolved. A given configuration of systems in the cloud may not be able to scale to developing workloads. Sharding, replication, batching, and similar improvements will be necessary to incorporate over time. Next, online software updates will be necessary for bug fixes, new features, enhanced security, and so on.

We describe the design and initial implementation of *Ovid*, a container-based framework for building, maintaining, and evolving distributed systems that run in the cloud. Each container runs one or more *agents* that communicate by exchanging messages. An agent is a self-contained state machine that transitions in response to messages it receives and may produce output messages for other agents. Initially, a system designer does not worry about scaling, failure handling, and so on, and develops relatively simple agents to implement a particular service. Next, we apply automatic *transformations* to agents such as replication or sharding. Ovid provides a suite of such transformations. Each transformation replaces an agent by one or more new agents and creates a refinement mapping [15] from the new agents to the original agent to demonstrate correctness. Transformations can be applied recursively, resulting in a tree of transformations.

When an agent is replaced by a set of agents, the question arises what happens to messages that are sent to the original agent. For this, each transformed agent has one or more *ingress agents* that receive such incoming messages. The routing is governed by routing tables: each agent has a routing table that specifies, for each destination address, what the destination agent is. Inspired by software-defined networks [9, 17], Ovid has a logically centralized controller, itself an agent, that determines the contents of the routing tables.

Besides routing, the controller determines where agents run and uses containerization to run agents as lightweight processes sharing resources. Agents may be co-located to reduce communication overhead, or run in different locations to benefit performance or failure independence. Ovid also supports on-the-fly reconfiguration. The result is what can be termed a "software-defined distributed system" in which a programmable controller manages a running system.

We have built a prototype implementation of Ovid that includes an interactive and visual tool for specifying and transforming distributed systems and a run-time environment that deploys and runs the agents in containers in the cloud. The interactive designer makes it relatively easy, even for novice users, to construct systems that are scalable and reliable. The designer can be run from any web browser. The run-time environment, currently only supporting agents and transformations written in Python, manages all execution and communication fully automatically. Eventually, Ovid is intended to support agents written in a variety of programming languages.

2 Design

A system in Ovid consists of a set of *agents* that communicate by exchanging messages. Each agent has a unique

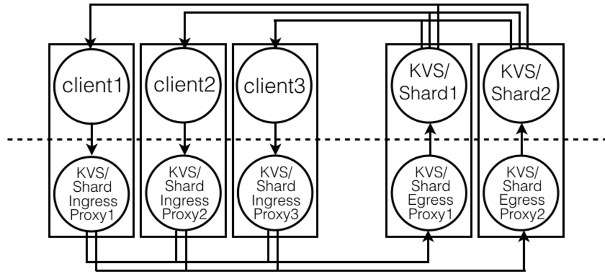

Figure 1: Boxes (rectangles) and agents (circles) for the key-value store that has been transformed to be sharded two-ways.

identifier and messages are sent to specific agent identifiers. When an agent receives a message, it updates its state and it may generate output messages that are sent to other agents. Every agent has a routing table that maps agent identifiers to agent identifiers.

For example, if a web server, acting as a client, uses a key-value store to store data, it can send `GET` and `PUT` requests to the key-value store using an agent identifier, say `'KVS'`. This means, the web server knows the key-value store as `'KVS'`. The routing table of the web server must then contain an entry that maps `'KVS'` to the unique agent identifier of the key-value store agent, and similarly, if the key-value store knows the web server as a `'client'`, the routing table of the key-value store agent must contain an entry that maps `'client'` to the identifier of the web server.

The agents can be *transformed*— replacing it with one or more new agents in such a way that the new agents collectively implement the same functionality as the original agent from the perspective of the other, unchanged, agents. In the context of a particular transformation, we call the original agent *virtual*, and the agents that result from the transformation *physical*.

For example, consider the key-value store used by the web server. We can *shard* the virtual agent by creating two physical copies of it, one responsible for all keys that satisfy some predicate $P(key)$, and the other responsible for the other keys. To glue everything together, we add additional physical agents: a collection of *ingress proxy agents*, one for each client, and two *egress proxy agents*, one for each key-value store agent.

An ingress proxy agent mimics the virtual agent `'KVS'` to its client, while an egress proxy agent mimics the client to a shard of the key-value store. The routing table of the client agent is modified to route messages to `'KVS'` to the ingress proxy.

Figure 1 illustrates the configuration as a directed graph and demonstrates the use of *layering* and *encapsulation*, common concepts in distributed systems and networking. Every physical agent is pictured as a separate node and agents that are co-located are shown in the same rectangle representing a box, which acts as a container. The directed edges between nodes illustrate the message traffic patterns between agents. Moreover, the configuration shows two layers with an abstraction boundary. The top layer shows an application and its clients. The bottom layer multiplexes and demultiplexes. This is similar to multiplexing in common network stacks. For example, the EtherType field in an Ethernet header, the protocol field in an IP header, and the destination port in a TCP header all specify what the next protocol is to handle the encapsulated payload. In our system, agent identifiers fulfill that role. Even if there are multiple layers of transformation, each layer would use, uniformly, an agent identifier for demultiplexing.

Note that, a transformation is essentially a special case of a refinement in which an *executable* specification of an agent is refined to another executable specification. Hence, in Ovid every transformation maps the state of the physical agents to the state of the virtual agent and the transitions of the physical agents to transitions in the virtual agent, in effect exhibiting the correctness of the transformation. Another paper on Ovid [4] includes examples of these refinements and focuses on the theory of how they work, while this paper presents the design and implementation of Ovid.

Ovid agents and transformations can be written in any programming language, as long as they conform to a specified line protocol (usually in the form of JSON records). The example specifically illustrates sharding, but there are many other kinds of transformations that can be applied in a similar fashion, among which:

- *State Machine Replication*: similar to sharding, this deploys multiple copies of the original agent. The proxies in this case run a replication protocol that ensures that all copies receive the same messages in the same order;
- *Primary-Backup Replication*: this can be applied to applications that keep state on a separate disk using read and write operations. In our model, such a disk is considered a separate agent. Fault-tolerance can be achieved by deploying multiple disk agents, one of which is considered primary and the others backups;
- *Load Balancing*: also similar to sharding, and particularly useful for stateless agents, a load balancing agent is an ingress proxy agent that spreads incoming messages to a collection of server agents;
- *Encryption, Compression, and Batching*: between any pair of agents, one can insert a pair of agents

that encode and decode sequences of messages respectively;

- *Monitoring, Auditing*: between any pair of agents, an agent can be inserted that counts or logs the messages that flow through it.

Above we have presented transformations on individual agents. In limited form, transformations can sometimes also be applied to sets of agents. For example, a *pipeline of agents* (in which the output of one agent form the input to the next) acts essentially as a single agent, and transformations that apply to a single agent can also be applied to pipelines. Some transformations, such as Nysiad [11], apply to particular configurations of agents. For simplicity, we focus here on transformations of individual agents only, but believe the techniques can be generalized more broadly.

Every agent in the system has a unique identifier. The agents that result from a transformation have identifiers that are based on the original agent's identifier, by adding new identifiers in a `'path name'` style. Thus an agent with identifier `'X/Y/Z'` is part of the implementation of agent `'X/Y'`, which itself is part of the implementation of agent `'X'`. In our running example, assume the identifier of the original key-value store is `'KVS'`. Then we can call its shards `'KVS/Shard1'` and `'KVS/Shard2'`. We can call the server proxies `'KVS/ShardEgressProxy`*X*`'`, and we can call the client proxies `'KVS/ShardIngressProxy`*X*`'` for some *X*.

The client agent in this example still sends messages to agent identifier `'KVS'`, but due to transformation the original `'KVS'` agent no longer exists physically. The client's routing table maps agent identifier `'KVS'` to `'KVS/ShardIngressProxy`*X*`'` for some *X*. Agent `'KVS/ShardIngressProxy`*X*`'` encapsulates the received message and sends it to agent identifier `'KVS/ShardEgressProxy1'` or `'KVS/ShardEgressProxy2'` depending on the hash function. Assuming those proxy agents have not been transformed themselves, there is again a one-to-one mapping to corresponding agent identifiers. Each egress proxy ends up sending to agent identifier `'KVS'`. Agent identifier `'KVS'` is mapped to agent `'KVS/Shard1'` at agent `'KVS/ShardEgressProxy1'` and to agent `'KVS/Shard2'` at agent `'KVS/ShardEgressProxy2'`. Note that if identifier X in a routing table is mapped to an identifier Y, it is always the case that X is a prefix of Y (and is identical to Y in the case the agent has not been refined).

Given the original system and the transformations that have been applied, it is always possible to determine the destination agent for a message sent by a particular source agent to a particular agent identifier.

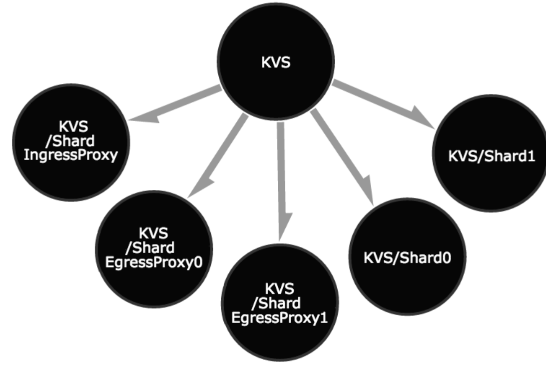

Figure 2: The Logical Agent Transformation Tree (LATT) for the two-way sharded `'KVS'` key-value store generated by the visualizer.

We represent a system and its transformation in a *Logical Agent Transformation Tree* (LATT). A LATT is a directed tree of agents. The root of this tree is a virtual agent that we call the *System Agent*. The "children" of this root are the agents before transformation. Each transformation of an agent (or set of agents) then results in a collection of children for the corresponding node.

A deployed system evolves. Ovid supports on-the-fly reconfiguration based on "wedging" agents [6, 2]. By wedging an agent, it can no longer make transitions, allowing its state to be captured and re-instantiated for a new agent. Brevity prevents us from going into detail how the state is sharded and transferred, but the refinement mapping provides the information about how the state of the transformed agent has to map back to the original agent. By updating routing tables the reconfiguration can be completed.

Eventually we would like these transformations to be determined and updated automatically based on the workload of the application and the environment in which it runs. In the current system, transformations are manually selected and updated by the user with the help of a visualizer component.

3 Implementation

Our implementation is comprised of two main parts: *The designer* creates the LATTs that represent the desired distributed system and transforms these LATTs creating a configuration that can be deployed. *Ovid core* then takes the configuration generated by the designer, builds the LATT as a real system with a collection of agent processes and deploys it in a given setting.

The designer models a given distributed system in the form of a LATT, and applies transformations on this LATT. It is implemented in JavaScript, to make it easy

to run it in a web browser, with 1200 lines of code. The designer has a visualizer component that can draw the LATT for a given system in the form of a graph. Users can select the transformations to be applied to a given system and these transformations create a new LATT that represents the new distributed system, as a combination of agents.

Using the LATT visualizer, the user can see how transformations change the LATT of a given system and which components are created as a result of the transformation. The corresponding graph for the LATT is kept up-to-date automatically. The visualizer also shows the high-level specifications of systems in the environment and how they are connected to each other. Using this information, the user can see how transformations affect the high-level specification of a system and easily understand how a given system interacts with other systems.

When a system is transformed only the LATT of that system is affected, but when it comes to deployment, other systems that are connected to the given system may be affected as well. The transformation function updates the deployment information for the environment once the LATT of a given system is updated. For example, Figure 2 shows the LATT generated by the visualizer for the two-way sharded key-value store we used as our running example. During this transformation, the LATTs of the clients that are connected to the `'KVS'` system are not transformed, but the deployment information of the clients are transformed to include the ingress proxies to be able to send their requests to the correct shard. The designer then creates a configuration that can be directly deployed by Ovid core.

Ovid core builds and deploys a distributed system created by the designer in a cloud environment. The current version of Ovid core is implemented with 5000 lines of Python code. Agents run as Python threads inside a *box*, which acts as a container, that is, an execution environment for agents. Eventually, we want to support agents written in any programming language running in containers maintained by platforms such as Docker [18] or Kubernetes [1].

For each agent, the box keeps track of the agent's attributes and runs transitions for messages that have arrived. In addition, each box runs a *box manager agent* that supports management operations such as starting a new agent or updating the Agent Routing Table (ART) of a given agent.

Boxes also implement the reliable transport of messages between agents. A box is responsible for making sure that the set of output messages of an agent running on the box is transferred to the sets of input messages of the destination agents. For each destination agent, this assumes there is an entry for the message's agent identifier in the source agent's routing table, and the box also needs to know how to map the destination agent identifier to one or more network addresses of the box that runs the destination agent. A box maintains the Box Routing Table (BRT) for this purpose.

Co-locating agents in a box saves message overhead, since co-located agents can communicate efficiently via shared memory instead of message passing. Currently a small PUT request takes on average 0.29 milliseconds when the client is co-located with the server and 1.38 milliseconds when the client and server are running in different boxes on Linux machines that are connected using a 1 Gbit Ethernet.

All routing information is maintained by the *controller* including the list of boxes and their network addresses, the list of agents and in which boxes they are running, and the agent identifier mappings used to route messages.

As in other software-defined architectures, we deploy a logically centralized controller for administration. The controller itself is just another agent, and has identifier `'controller'`. The controller agent itself can be transformed by replication, sharding and so on. For scale, the agent may also be hierarchically structured. Depending on the deployment of the system itself, the controller itself can be transformed and deployed accordingly, for instance deployed across a WAN. However, here we assume for simplicity that the controller agent is physically centralized and runs on a specific box.

As a starting point, the controller is configured with the LATT, as well as the identifiers of the box managers on those boxes. The BRT of the box in which the controller runs is configured with the network addresses of the box managers.

First, the controller sends a message to each box manager imparting the network addresses of the box in which the controller agent runs. Upon receipt, the box manager adds a mapping for the controller agent to its BRT. The controller can then instantiate the agents of the LATT, which is accomplished by the controller sending requests to the various box managers.

Initially, the agents' routing tables contain only the `'controller'` mapping. When an agent sends a message to a particular agent identifier, there is an "ART miss" event. On such an event, the agent ends up implicitly sending a request to the controller asking it to resolve the agent identifier to agent identifier binding. The controller uses the LATT to determine this binding and responds to the client with the destination agent identifier. The client then adds the mapping to its routing table.

Next, the box tries to deliver the message to the destination agent. To do this, the box looks up the destination agent identifier in its BRT, and may experience a "BRT miss". In this case, the box sends a request to the controller agent asking to resolve that binding as well. The destination agent may be within the same box as the

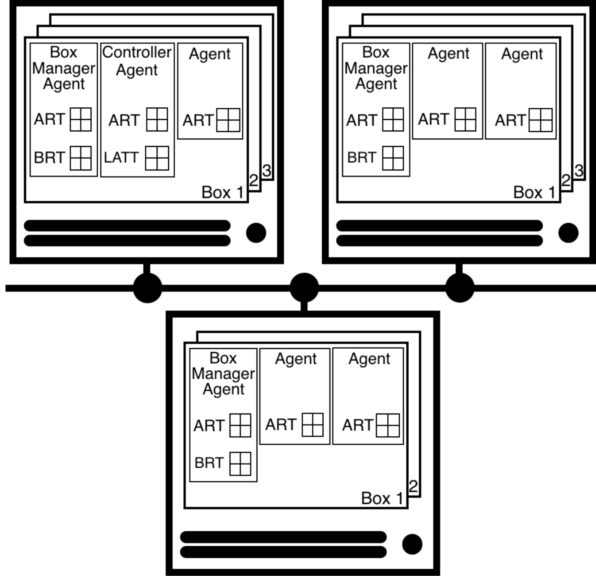

Figure 3: The physical deployment of a system in the cloud. Multiple agents can run in a box and multiple boxes can run on machines that are connected to each other through the network.

source agent, but this can only be learned from the controller. One may think of routing tables as caches for the routes that the controller decides.

Figure 3 shows a possible layout of systems created using the Ovid framework. There are three physical machines connected through the network. Machines can run multiple boxes, acting as containers, which may run multiple agents themselves.

Ovid core is implemented in an object-oriented manner, where every agent in the system, including the box managers and the controller, extends from an Agent class. Moreover, each transformation and the agents that implement it is created as a separate module, and each module extends from a base module. The base module includes the base agent implementation and the underlying messaging layer as well as other utility libraries. On top of this base module, every module includes transformation specific agents such as the ingress and egress proxies. Ovid is designed in an object-oriented and modular fashion to be easily evolvable itself. New transformations can be added to Ovid by adding new modules that are independent of existing modules.

4 Related Work

There has been much work on support for developing distributed systems as well as deploying long-lived distributed systems that can adapt to evolving environments.

Mace [13] is a language-based solution to automatically generate complex distributed system deployments using high-level language constructs. Orleans [8] and Sapphire [22] offer distributed programming platforms to simplify programming distributed applications that run in cloud environments.

In [21], Wilcox et al. present a framework called Verdi for implementing practical fault-tolerant distributed systems and then formally verifying that the implementations meet their specifications. Systems like Verdi can be used in combination with Ovid to build provably correct large-scale infrastructure services that comprise multiple distributed systems. These systems can be employed to prove the safety and liveness properties of different modules in Ovid, as well as the distributed systems that are transformed by Ovid.

Another approach to implementing evolvable distributed systems is building reconfigurable systems. Reconfigurable distributed systems [5, 7, 12, 14] support the replacement of their sub-systems. In [3], Ajmani et al. propose automatically upgrading the software of long-lived, highly-available distributed systems gradually, supporting multi-version systems. Horus [20, 16] and Ensemble [10, 19] employ a modular approach to building distributed systems using *micro-protocols* that can be combined together to create protocols.

5 Conclusion and Future Work

Ovid is a software-defined distributed systems framework for building large-scale distributed systems that have to support evolution in the cloud. Using Ovid, a user designs, builds and deploys distributed systems comprised of agents that run in containers. Our prototype implementation of Ovid supports agents and transformations of these agents written in Python. Next, we will support agents written in any programming language running in containers. We also want to develop tools to verify performance and reliability objectives of a deployment. We then plan to do evaluations of various large-scale distributed systems developed using Ovid. Eventually, Ovid will be able to create complex distributed infrastructure services as a combination of systems running in containers in the cloud.

Acknowledgments

The authors are supported in part by AFOSR grants FA2386-12-1-3008, F9550-06-0019, FA9550-11-1-0137, by NSF grants CNS-1601879, 0430161, 0964409, 1040689, 1047540, 1518779, 1561209, 1601879, CCF-0424422, by ONR grants N00014-01-1-0968, N00014-09-1-0652, by DARPA grants FA8750-10-2-0238, FA8750-11-2-0256, and by gifts from Yahoo!, Microsoft Corporation, Infosys, Google, Facebook Inc.

References

[1] Kubernetes. `http://kubernetes.io/`. Accessed March 7, 2016.

[2] ABU-LIBDEH, H., VAN RENESSE, R., AND VIGFUSSON, Y. Leveraging sharding in the design of scalable replication protocols. In *Proceedings of the Symposium on Cloud Computing* (Farmington, PA, USA, October 2013), SoCC '13.

[3] AJMANI, S., LISKOV, B., AND SHRIRA, L. Modular software upgrades for distributed systems. In *Proceedings of the 20th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2006), ECOOP'06, Springer-Verlag, pp. 452–476.

[4] ALTINBUKEN, D., AND VAN RENESSE, R. Ovid: A software-defined distributed systems framework to support consistency and change. *IEEE Data Engineering Bulletin* (Mar. 2016).

[5] BIDAN, C., ISSARNY, V., SARIDAKIS, T., AND ZARRAS, A. A dynamic reconfiguration service for corba. In *4th International Conference on Distributed Computing Systems* (1998), ICCDS'98, IEEE Computer Society Press, pp. 35–42.

[6] BIRMAN, K., MALKHI, D., AND VAN RENESSE, R. Virtually synchronous methodology for dynamic service replication. Tech. Rep. MSR-TR-2010-151, Microsoft Research, 2010.

[7] BLOOM, T. Dynamic module replacement in a distributed programming system. Tech. Rep. MIT-LCSTR-303, MIT, 1983.

[8] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J., PANDYA, R., AND THELIN, J. Orleans: Cloud computing for everyone. In *ACM Symposium on Cloud Computing (SOCC '11)* (October 2011), ACM.

[9] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 1–12.

[10] HAYDEN, M. G. *The Ensemble System*. PhD thesis, Cornell University, Ithaca, NY, USA, 1998. AAI9818467.

[11] HO, C., VAN RENESSE, R., BICKFORD, M., AND DOLEV, D. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI '08, USENIX Association, pp. 175–188.

[12] HOFMEISTER, C. R., AND PURTILO, J. M. A framework for dynamic reconfiguration of distributed programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems* (1991), ICDCS 1991, pp. 560–571.

[13] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 179–188.

[14] KRAMER, J., AND MAGEE, J. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering 16*, 11 (November 1990), 1293–1306.

[15] LAMPORT, L. Specifying concurrent program modules. *Transactions on Programming Languages and Systems 5*, 2 (April 1983), 190–222.

[16] LIU, X., KREITZ, C., VAN RENESSE, R., HICKEY, J., HAYDEN, M., BIRMAN, K., AND CONSTABLE, R. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating System Principles* (Kiawah Island Resort, SC, USA, December 1999).

[17] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (March 2008), 69–74.

[18] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (March 2014).

[19] VAN RENESSE, R., BIRMAN, K. P., HAYDEN, M., VAYSBURD, A., AND KARR, D. Building adaptive systems using ensemble. *Software Practice and Experience 28*, 9 (August 1998), 963–979.

[20] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A flexible group communication

system. *Communications of the ACM 39*, 4 (April 1996), 76–83.

[21] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed system. In *Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation* (Portland, OR, USA, June 2015), *Implementation* (Portland, OR, USA, June 2015), PLDI 2015.

[22] ZHANG, I., SZEKERES, A., AKEN, D. V., ACKERMAN, I., GRIBBLE, S. D., KRISHNAMURTHY, A., AND LEVY, H. M. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 97–112.