2018 IEEE Symposium on Service-Oriented System Engineering

If Docker Is The Answer, What Is The Question?

A Case for Software Engineering Paradigm Shift Towards Service Agent Orientation

Hong Zhu and Ian Bayley
School of Engineering, Computing and Mathematics
Oxford Brookes University, Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

***Abstract*—The recent rise of cloud computing poses serious challenges for software engineering because it adds complexity not only to the platform and infrastructure, but to the software too. The demands on system scalability, performance and reliability are ever increasing. Industry solutions with widespread adoption include the microservices architecture, the container technology and the DevOps methodology. These approaches have changed software engineering practice in such a profound way that we argue that it is becoming a paradigm shift. In this paper, we examine the current support of programming languages for the key concepts behind the change in software engineering practice and argue that a novel programming language is required to support the new paradigm. We report a new programming language CAOPLE and its associated Integrated DevOps Environment CIDE and demonstrate the utility of both.**

***Keywords*—** Cloud computing, Microservices, DevOps, Service agent orientation, Software engineering paradigms, Parallel and distributed programming models, Software development methodology, Programming languages, Integrated Software Development Environment.

I. Introduction

Cloud-based applications are becoming more and more complex, whilst having to meet unprecedented and ever increasing demands on system performance, scalability, reliability and maintainability. Solutions for meeting this demand have been proposed that increase system flexibility by means of greater elasticity and evolvability. These solutions include the microservices architecture [1, 2, 3], container technology [4, 5], DevOps tools and methodology [6], etc. Behind these solutions is a set of novel concepts that have become the basis for a set of new techniques. In this paper we will argue that the changes to practice that they bring about are so fundamental that they are causing a *paradigm shift* right now. We will recognize the key characteristics of the new paradigm, identify the missing pieces in that emerging paradigm, and propose further research directions. We will also report our own research and demonstrate how the power of new paradigm can be further strengthened.

The remainder of this paper is organized as follows. Section II discusses what is meant by a software engineering paradigm and why paradigm shifts become necessary. Section III reviews the current best practice in software engineering of cloud native software to identify the characteristic features of the emerging new paradigm. Section IV examines existing programming models in the light of the new paradigm by comparing our service agent model to actor and reactive programming models. Section V reports our ongoing research into the development of a new programming language called CAOPLE, and an associated DevOps environment called CIDE. Section VI concludes the paper with a summary and a discussion of further research.

II. Software Engineering Paradigms

A. What is a paradigm?

A paradigm of software engineering is a consistent set of software development techniques and methodologies guided by a philosophical model of computing; this is an abstract model of computer systems and of software systems running on hardware. The model dictates how applications should be constructed and how they should evolve.

For structured software engineering, the first well-established paradigm, the philosophical model can be summarized as: *computing is processing of data stored in the computer*. The hardware is assumed to be a stand-alone general-purpose digital mainframe computer with a collection of data storage and input/output devices. A software system is considered to be a collection of procedures, each defining a routine operation in the processing of data, and organized in a hierarchical structure with a top-level "main" procedure for overall control.

The philosophical model for object-oriented software engineering, on the other hand, can be summarized as: *computing is interactions between objects, which are computational entities that encapsulate data and operations*. The hardware can be a network of computers instead of simply a standalone. Data is no longer separated from the code that processes it.

Note that the existence of a philosophical model is essential for a paradigm to become well-established. This is even true for the paradigms that have not yet become mainstream. For example, logic programming views computing as *logical inference* whereas functional programming views it as *function application*, in the mathematical sense of the symbol manipulation in lambda calculus [7].

Three conditions are needed for a paradigm to become mainstream. First of all, the philosophical model must be supported directly by the hardware and enable the power of the hardware to be fully utilized. Secondly, there should be an associated development process such as the waterfall method for structured programming and the use case driven and agile process models for the object-oriented paradigm. Finally, there should be an associated programming language based on the philosophical model. Examples include the languages Fortran, Basic, Pascal and C for structured programming and the languages Smalltalk, Eiffel, C++ and Java for object-orientation.

B. *What drives paradigm shifts?*

Paradigms guide, but also impose constraints, on how we develop, operate, maintain and evolve computer applications. When hardware advances make new kinds of application possible, these constraints can become a development bottleneck. When that happens, the philosophical model of computing needs to change, in order to improve productivity and software quality. This is called a *paradigm shift*, a concept due to the American physicist and philosopher Thomas Kuhn who defined it as a fundamental change in the basic concepts and experimental practices of a scientific discipline [8].

The two major paradigm shifts in software engineering over the last few decades have been from assembly code to structured software engineering, using procedural high-level programming languages, and from that to object-oriented software engineering. The main driving force behind both paradigm shifts was a desire to improve software productivity and reliability and to make it possible to write more complex and larger-scale software systems.

Efficiency, in contrast, has always been a lesser concern. In fact, both paradigm shifts were at the expense of efficiency. The intention in both cases was that the programmer should be able work at a higher level of abstraction, concentrating on the business logic of the application, with the compiler mapping a high-level model of computation used by the programmer to the low-level model of the machine, with some performance penalty.

It is the advances in hardware, making computing cheaper, faster and smaller, that prompts the aforementioned desire for more complex and larger-scale systems. The limitations of the old paradigm became a bottleneck that the paradigm shift then overcomes. With this in mind, it is instructive to note that the cheaper/faster/smaller trend has continued, bringing about wireless networks and smart devices. These have led to the development of cloud, mobile, and IoT (Internet of Things) applications.

However, the object-oriented paradigm, in which computer systems are viewed as consisting of passive objects waiting for method calls from each another, is a poor fit for such applications, in which the computational entities are autonomous, collaborative and proactive. The programmer must therefore deal with all the technical details of network communication, collaboration protocols and fault tolerance. Even to deploy a software system to a cluster and to monitor its execution is a non-trivial task. These technicalities could instead become only a compiler concern if the move was made to a yet higher level of abstraction with another paradigm shift.

III. Review of The Current Practice

We will now deduce an ideal philosophical model for cloud computing in the proposed new paradigm by reviewing the dominant software architectures and platforms for purpose-built or *cloud-native* applications. We will also in this section review the development process models from a management perspective in order to understand what are the bottlenecks of the existing paradigm. This will lay a foundation for a review in the next section of current programming languages. That review will help to identify a route from the current state-of-the-art to a more mature paradigm for cloud computing that supports the philosophical model.

A. *Microservices*

The microservices architecture became widely adopted for cloud-native applications during the 2010s. Santoli [9] pointed out that all successful IT companies have taken an aggressive approach to adopting it. Well-known examples include NetFlix [10, 11], Amazon [12], EBay [13], Google [13] and Microsoft (with Azure) [14]. A global survey by Smartbear in 2016 [15] found that 73% of organizations provide both internal and external APIs, which is a key technique used to integrate services in the microservices architecture [16].

(1) The Concept of Microservices

As Martin Fowler [1] puts it, the idea of the microservices architectural style is that an application consists of "a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API". These services are "independently deployed" with a "bare minimum of centralised management". Nevertheless, the exact definition is still a matter of controversy [17, 18].

Focusing on the software architecture point of view, the microservices architectural style has the following properties:

- Components are services.
  - Each component is *autonomous*, i.e. running on its own process and managing its own resources.
  - Each implements a single function and so is of *fine granularity*.
  - Each can be *independently deployed* to different machines over a cluster.
- Connectors are service requests and responses.
  - Communication is only through *service requests and responses* via a lightweight mechanism.
  - Connections between a service provider and a service requester can be established dynamically at runtime.
- Configuration is dynamic and decentralised.
  - Services communicate with each other to form a collaborating network, typically without a central controller.
  - New copies of a service can be created if needed and idle existing copies can be destroyed, both at runtime.
  - Multiple copies of a service may exist in the system and they can be distributed to multiple machines.

(2) Benefits of The Microservices Architecture

The benefits of microservices for large cloud-based applications [19] over alternative styles, such as monolithic architecture, include the following:

- *Continuous software evolution*. Introducing new functionality can be achieved by adding new services, and bugs can be fixed at runtime by replacing existing services. Only one service (and possibly its dependencies) needs to be rebuilt and redeployed. This can be done without stopping the rest of the application, which is of particular importance for cloud-based applications.
- *Seamless technology integration*. It is easy to combine many different programming languages, varieties of database, hardware and software environments and other computing technology depending on what fits best, all running on a heterogeneous cluster of different platforms.
- *Optimal runtime performance*. This can easily be achieved in the microservices architecture by running multiple copies of a service when there is high demand and balancing the system load by deploying the right number of these copies to the correct servers, and moving them between servers.
- *Horizontal scalability.* The system can be easily scale out by running multiple copies of individual services on new servers in response to demand rather than running multiple copies of the whole system.
- *Reliability through fault tolerance*. Fault tolerance can be achieved by running multiple copies of some services for redundancy and multiple implementations of the same service for diversity. Recovery from failure simply requires a new copy of the service.

(3) Challenging Problems

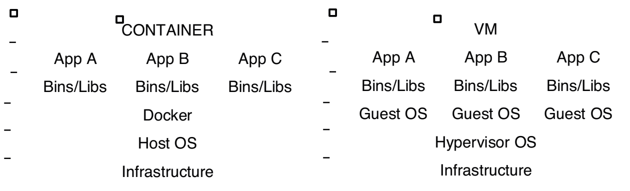
Adopting the microservices architecture is difficult, however, because of the following problems.

- *Complexity in Deployment*, due to the large number of services that must be deployed to the cluster and started quickly. This deployment must be done dynamically to achieve the advantages of load balance and elastic scalability. As Daya *et al.* pointed out [20], "*microservices cause an explosion of moving parts. It is not a good idea to attempt to implement microservices without serious deployment and monitoring automation*".
- *The Need to Monitor Execution*, to diagnose hardware and software failures quickly and to replace failed parts with new instances, thereby conferring the benefits of fault tolerance and reliability. Workload must also be monitored to achieve load balance and elastic scalability. Monitoring services is even more difficult when there is a large number of them, due to the fine granularity, all running in parallel in a distributed environment.
- *Network Latency*, which is a greater problem when services communicate with each other a lot over the network.
- *Cognitive Load* of the extra complexity brought by the microservices architecture including message formats, load balance and fault tolerance. This is shifted to the monitoring tools. The usual problems of complex parallel and distributed software remain. One approach to developing the microservices architecture is to refactor a monolithic architecture in a gradual way. This approach has worked for eBay, Twitter, Google, and Amazon [13].

These challenges have led to the rise of technologies and methodologies that support microservices such as container technology and DevOps, both of which we review in the next two subsections.

B. Container Technology

Container technology [21, 22] enables a piece of runnable software code to be wrapped, together with any resources needed, into a package, called a *container image*. This is then deployed onto a machine, which generates a *container instance* running as an isolated process in user space. Each machine can run several containers, all sharing the same OS kernel. Each container typically takes up far less space than a virtual machine would (e.g. tens of MBs vs several GBs). It can therefore be sent through a network more quickly and started almost instantly as a process on an operating system, whereas it takes far longer, often a few minutes, to reboot a virtual machine. Containers also provide separation between users, thereby achieving the same security and privacy advantages that virtual machines have. Figure 1 shows the differences between container and virtual machine techniques [23].



(a) Container (b) Virtual Machine

Figure 1. Comparison of Container with Virtual Machine [23]

Since a container comes with all the resources it needs to run, it can be deployed on any machine that runs the operating system it targets. This is described as "*package once and deploy anywhere*" [24] but that differs from "*write once and run anywhere*" motto of Java because a container can only run on one operating system. The code must be packaged once for each operating system. Further flexibility is given by the fact that containers can be run on virtual machines which can then be on different platforms, as shown in Figure 2.

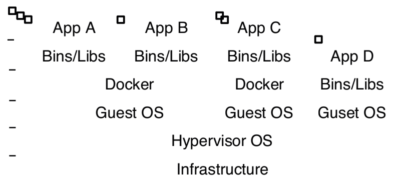


Figure 2. Containers and Virtual Machines Used Together [23]

Launched in 2013, Docker is the *de facto* industry standard for container technology, with 40% of enterprises using it and 30% more planning to do so. Figure 3 charts this rapid growth in popularity, counting pulls from GitHub [23]. Docker is also provided as container-as-a-service, as seen in AWS ECS (35%), Azure Container Service (11%), and Google Container Engine (8%). Docker combines two open standards: (a) Docker Image Specification, which defines the format used to package contents into a container and (b) Docker Runtime Specification, which defines the runtime components.

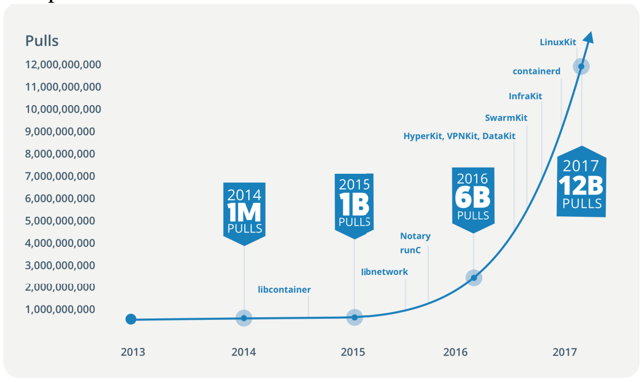


Figure 3. The Growth of Popularity of Docker [23]

Because the runtime environment of a service is packaged with the code, there is no need to configure hardware and software, nor versions of languages and tools. The complexity is pushed into containers that are easy to build, share and run.

The deployment of applications can be automated with the use of container orchestration engines, which deploy a suite of containers to a cluster of machines in a pre-scripted configuration. Docker has its own built-in orchestration engine called Swarm but alternatives include Kubernetes and Mesos [25].

C. DevOps

DevOps, a concatenation of (software) development and (IT) operations, is a whole life cycle methodology that stresses the integration of those two tasks, which are traditionally handled by separate teams. This integration requires the removal of communication boundaries and must happen as early as possible for the greatest gains at combating complexity. DevOps requires that the products of development be moved smoothly in a pipeline in turn to platforms for development, testing, stage and operation. Each of these is typically a heterogeneous cluster of computers. Figure 4 shows a typical DevOps pipeline as suggested by Sharma and Coyne [6].

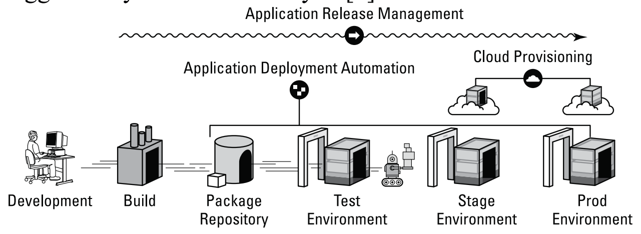


Figure 4. A Typical DevOps Pipeline [6]

Many tools are available for each of these stages, in addition to the traditional phases of software development, but there is no single tool available for them all [26]. DevOps tools fall into the following categories.

- *Software Package Management*: package creation, artifact repositories, and staging prior to deployment. Docker is also referred to as a DevOps tool because it provides software packaging as well as automated container deployment and metrics for monitoring the execution of containers.
- *Service Release Management*: change management, release approval, and release automation.
- *System Configuration and Deployment Management*: Infrastructure and deployment to a cluster, for example, Jenkins, Puppet, Vagrant, Ansible, etc.
- *System Monitoring*: Collecting system state data, statistical analysis of that data and visual display, for example, Nagios/Icinga, Monit, Collectd/Collectl.
- *Log File Analysis*: Used for diagnostic purposes, such as ELK (Elasticsearch, Logstash and Kibana).
- *Service registration and discovery*: Registration of and access to services deployed to a cluster, for example, Consul, Zookeeper, etcd,

DevOps applies many of the principles of agile methodology to large-scale clusters and due to its widespread popularity it is often used with microservices and container technology. The overall DevOps adoption rate rose from 66% in 2015 to 74% in 2016 to 78% in 2017. For larger enterprise organizations, the adoption rates are even higher: 81% in 2016 and 84% in 2017 [27, 28].

D. Discussion

Cloud-native applications are made not from objects and classes but from microservices, which are active computational entities dynamically created from a template (container image) and deployed to a network of computers. The communication links are dynamically bound. Systems evolution is by continuous integration and continuous delivery. A collection of communicating microservices forms an ecosystem. Likewise, the teams working on each microservice also form an ecosystem.

These changes have permeated all aspects of software engineering and the new practices and techniques have evolved to replace them. This is why we say a software paradigm shift is taking place right now.

IV. Search for New Programming Languages

A crucial aspect of the emerging new paradigm as yet not discussed is the choice of programming languages in which microservices are written. Many new programming languages have arisen in the past few years. Some like ActorScript [29] are specifically aimed at cloud-based applications while others, like Go [30] and Scala [31], are general purpose languages with an emphasis on network systems. A thorough survey is beyond the scope of this paper, so here, we discuss a few general approaches.

A. Actor Model

The Actor model, a mathematically-based formalism dating back to a paper from 1973 [32], proposes actors as universal primitives of concurrent computation. Each actor is an entity that can do the following:

a) Send messages to other actors;
b) Perform computation in response to messages it receives;
c) Create new actors either locally or remotely.

An actor A may modify its own private state but it cannot directly change that of another actor B, though B may choose to change its state in response to a message from A.

Messages sent between actors must be:

- Sent *directly* to the receiver using a unique reference to the receiver known to the sender;
- *Location transparent*, meaning that the sender only needs the receiver's reference but not its location;
- *Asynchronous*, meaning that the sender is not blocked waiting for delivery of the message.

A sender A can obtain the unique reference to a receiver B in one of the following ways:

- *Initial condition*: B is one of a number of fixed actors in the system environment known to all actors in the system.
- *Parenthood*: When A creates B, A obtains the unique reference to B.
- *Endowment*: When A is created, its parent passes it the reference to B.
- *Introduction*: Another actor C has sent A the reference to B.

The above so-called *reference capability model* is identical to that of object-orientation and is too restrictive for service-oriented systems. For this reason, Akka adds to these four the ability for A to search for the reference to B. Akka is based on the so-called *supervision hierarchy* structure of actor systems. If an actor A creates actor B, then A is the parent of B in the hierarchy and B is one of its children. As shown in Figure 5, the hierarchy has three guardian nodes as follows.

- *User Guardian*, representing the user, and which can create multiple actor systems.
- *System Guardian*, which creates all the System's internal actors.
- *Root Guardian*, which creates them both.

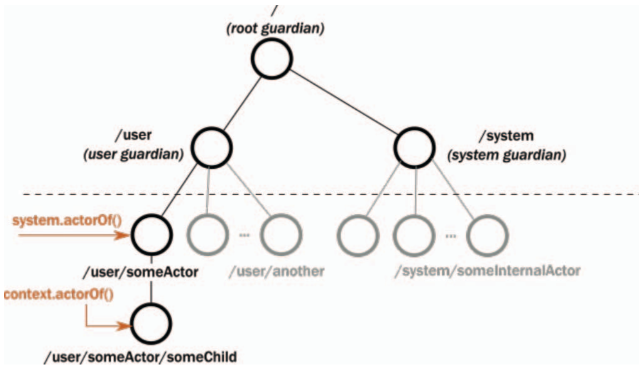


Figure 5. Akka's Supervision Hierarchy of Actors [33]

An actor can then be searched for from some point in the hierarchy, using an Akka method like *actorSelection*(…), and often using a wildcard.

The supervision hierarchy also enforces the management responsibility of actors in that a parent is responsible for terminating its children and dealing with their failures. An actor has four lifecycle methods, each of which can be overridden.

- *preStart*(): invoked after the actor has been created and just before the actor is started;
- *postStop*(): invoked just after the actor is stopped;
- *preRestart*(): invoked just before a failed child actor is restarted, as might be done as part of the actor's failure management strategy;
- *postRestart*(): invoked just after the restart, enabling re-initialization of the actor.

It is possible to conceive of a pure actor system in which everything is an actor, including even primitive data types such as integers, real numbers and strings, but no practical programming language exists for this as far as we know. More usually, an actor is a kind of system component, such as a web service, obtained by adding an Akka library to an existing programming language such as Scala or Java [33]. More than 50 other such libraries are listed in the Wikipedia page for Actor model [34].

An alternative is to extend an existing programming language with language facilities for supporting actors, as seen with ActorScript [**Error! Bookmark not defined.**]. More than 20 programming languages claim to support the actor model [34], but Akka is perhaps the most mature implementation of the actor model.

B. Reactive Model

The reactive model is event-driven whereas the actor model is message-driven; both are asynchronous. The distinction between the two is that messages are directed to a clear single destination, whereas events are not: they are "facts for others to observe" [35].

A consequence of this is that in the reactive programming style, control flow is driven not by the thread of execution but by the availability of new information. Reactive programming languages therefore make it possible to specify what actions must be taken in response to state changes. These state changes are thereby automatically and efficiently propagated across the network of dependent computations by the underlying execution model. For this reason, the reactive model has become popular for concurrent programming, although its origins are in the dataflow declarative programming languages of the 1980s [36].

Like actor-based programming, reactive programming can be enabled with either a language extension or a new library. Of the dozen such approaches recorded in a survey published in 2013 [36], the ReactiveX library for cloud computing has received particular attention from industry.

ReactiveX provides operators for declaratively composing sequences of data and events while abstracting away from concerns such as low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O. Such operators include those to filter, select, transform, combine or compose sequences and resemble those of functional programming languages.

ReactiveX is polyglot in the sense that it has been extended to many (18 so far) languages, giving for example RxJava as used by Netflix to make their entire service layer asynchronous, RxPython and RxJS (for JavaScript); it has also been extended to three platforms.

ReactiveX provides a framework based on the Observer design pattern. An abstract class *Observable* (the Subject class in the Observer pattern) may emit any number of items (including zero) and then terminate either successfully or with an error. Another abstract class *Subscriber* (the Observer class in Observer pattern) reacts to this sequence in a manner specified in a concrete subclass by overriding three abstract methods:

- *onNext*(), invoked when it receives an item;
- *onComplete*(), invoked when it terminates successfully;
- *onError*(), invoked when it fails with an error.

A subscriber links to an Observable object by invoking the latter's method *subscribe*(*Subscriber*: *sub*), obtaining a Subscription object that can then later be cancelled. The Observable then invokes *onNext*(), *onComplete*() and *onError*() when appropriate.

In the Android platform version of ReactiveX, the Observables and Subscribers may be placed on different threads, by using different arguments in two method calls *observerOn*(*scheduler*) and *subscriberOn*(*scheduler*).

C. Service Agent Model

As a part of software engineering methodology for internet-based applications, in 2000 we proposed an agent-oriented parallel computing model [37, 38], which turns out to be a good fit for cloud applications in the microservices architecture. Here, agent means service provider, as in estate agent and travel agent. Although our notion of agent was inspired by the notion of agent in distributed artificial intelligence, it is different from that in the agent models proposed and advanced in the AI community. Each agent is a computational entity that is:

- *Active*, running on its own process, and distributed to a network of computers, and
- *Autonomous*, in that only the agent itself can change its own state and it decides which actions to perform and when.

An agent encapsulates the following elements:

- *A set of state variables*, each of which can either be visible to other agents outside or invisible to them and known only by the agent itself.
- *A set of actions* the agent can perform, each of which on termination generates an event that will be seen by other agents if the action is visible to them
- *A description of the environment*, which lists which other agents are being observed for their visible actions and states.
- *A set of behavior rules*, defining how the agents change their state and take action in response to external events and changes in internal conditions.

Another crucial concept of our model is *caste*, which acts as the classifier for agents in much the same way as class is the classifier for objects. Each caste can therefore be thought of as a template from which agents are instantiated and created. Similarly, a caste can inherit from another caste. However, whereas an object's membership of a class is fixed at compile time, an agent can join, quit, suspend and resume its membership to a caste dynamically, thereby demonstrating adaptive behavior.

Here is an example of caste in the CAOPLE programming language, which is based on the service agent model. Please see Section V for more information about CAOPLE.

```
CASTE Chatter(givenName: STRING){
   OBSERVE Chatter;
   VAR name: STRING;
   ACTION Say(word: STRING) { }
   INIT{
         name:=givenName;
         Say("Hello, World!") ;
   }
   BODY{
      WHEN EXIST x in Chatter:Say("Hello, World!"){
         Say("Hello, ” + x.name);
      }
   }
}
```

The OBSERVE clause indicates that each agent of caste `Chatter` observes all other agents of the caste. The INIT clause is the list of statements executed when the agent is created; these save parameter `givenName` into visible state variable `name` and perform the action `Say(“Hello, World!”)`. The BODY clause, which is repeatedly executed until the agent terminates, responds to any such action from another agent `x` with the action `Say(“Hello, ” + x.name)`, where `x.name` is the name of `x`.

The communication mechanism here, in which one agent takes visible actions while another observes, is fundamentally different from the subscribe-publish mechanism because agents can be created, deployed over a network, and destroyed dynamically. This is a close match with the needs of microservices architecture, where a service must communicate with multiple copies of other services that are dynamically created and deployed to multiple machines in a cluster. In spite of this flexibility, strong type checking can be performed statically.

A further virtue of this mechanism, in addition to its simplicity, is that communication is location transparent and at a high level of abstraction; the programmer does not need to know which agents are in the system, nor where they are in the network, nor any details about communication ports or low-level synchronization primitives. Furthermore, when an agent's action is observed, its identity can be obtained if needed, as is done with agent `x` above. This breaks the reference capability limitation.

There is strong support in the service agent model for code deployment to a remote machine. For example, the following statements deploy two agents of caste `Chatter` to two different machines.

```
CREATE Chatter(“John”) @ "192.168.1.65";
CREATE Chatter(“Peter”) @ "192.168.1.71";
```

Other deployment-related operations for changing a caste’s membership can be seen in Section V.A.

Table 1 summarises how the concepts and language facilities in the service agent model match the key concepts in the microservices architectural style.

Table 1. How Service Agent Model Supports MS

Concept of Microservices	**Meanings**	**Concept in Service Agent Model**
Service	The **functionality** provided by a computer system and delivered to the users	Service
	The **computational entity** that provides the services in the above sense	Agent
Microservice	**Identical copies** of a service, where each copy is a runtime computational entity	Agent
	A **template** from which instances can be generated and deployed to different servers	Caste

D. Summary

From the summary in Table 2, it appears that the service agent model is the most suitable for programming cloud applications in a microservices architecture.

Table 2. Comparison of Programming Models

	Actors	**Reactive**	**Service Agent**
Runtime element	Actors, [objects]	Observables, Subscribers, Objects	Agents
Program unit	Actor types, [Classes]	Sub-Observables, Sub-Subscribers, Class	Castes
Reference propagation model	Reference capability model	Reference capability model	Reference capability model + message sender identification
Uses of Reference propagation Model	Lifecycle management, Communication address	Lifecycle management, Subscription for communication	Lifecycle management [optional], Communication address [optional]
Communication mechanism	Asynchronous, Direct addressing	Asynchronous, Subscribe-publish, Explicit event-driven	Asynchronous, Implicit event-driven, Sender ID identifiable
System structure	Centrally organised hierarchy	Flat, No central organisation	No central organisation
Location transparency	Implicit by reference	No	Yes

V. THE EXPERIMENTS OF CAOPLE AND CIDE

The service agent model has been realized with a programming language COAPLE [39] and an integrated DevOps environment called CIDE for editing, compiling, deploying, executing and testing code in a cluster. This section examines each in turn and demonstrates their support for the new paradigm.

A. The CAOPLE Programming Language

CAOPLE has the following properties. It is:

- *Purely Agent-Oriented*, in contrast to some languages that allow classes alongside agents as an alternative kind of building block.
- *Caste-Centric*, in the sense that agents can only be created by instantiating castes, unlike other languages where agents can be coded directly.
- *Imperative*, so programs take the form of a sequence of commands, in contrast to some AI-inspired languages that define agent behaviors in a logic or rules-based approach, game-theory approaches that employ utility functions and other approaches that are based on an organizational/social model.

Network transparency and *write-once-run-anywhere* was achieved by compiling source code to target a virtual machine CAVM-2. The only configuration required is to install CAVM-2 on every node of a cluster.

(1) Example 1: Hello World

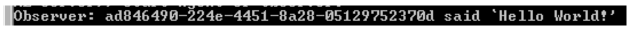
The following “Hello World” example illustrates the notion of caste, which is not only a compilation unit but also the unit for code deployment and execution. CIDE can be used to create instances of Peer and distribute them to machines on a cluster. Each execution of an agent of the caste Peer will perform an action `Say("Hello World!")`.

```
CASTE Peer(){
    ACTION Say(word: STRING) { }
    INIT{ Say("Hello World!"); }
    BODY{ }
}
```

To test whether this program behaves as we expect, instead of modifying the code, we can write another caste to observe its behaviour as follows.

```
USES Peer;
CASTE Observer() {
    OBSERVE Peer;
    INIT { }
    BODY {
        VAR word: STRING;
        WHEN EXIST x IN Peer: Say(RCV word) {
            print "Observer: "+ x.toString +
              " said ‘ " + word + “’”;
        }
    }
 }
```

Provided that the `Observer` agent is created before the `Peer` agent, when it is then executed, it will print the following message.



```
Observer: ad846490-224e-4451-8a28-05129752370d said ‘Hello World!’
```

where “`ad846`…” is the universally unique identifier of such an agent. The creation of agents can also be done programmatically as in the caste below that creates an `Observer` and two `Peers`.

```
USES Peer, Observer;
CASTE Builder2a(){
    INIT{
        CREATE Observer();
        wait 100;
        CREATE Peer() @ "192.168.1.65";
        CREATE Peer() @ "192.168.1.71";
    }
9:     BODY {}
10: }
```

The IP addresses are optional and if they are omitted then the agents can be located anywhere on the network. They need not be literals and can be constructed at run time. Note that the Observer agent can be run on a different machine where the Peer agents run.

(2) "Deployment" Mechanisms in CAOPLE

The following caste-membership operation statements form a rich set of "code deployment" operations for distributed programming.

```
AgentCreationStatement ::=
   create [AgentVar of] casteName ([params])
      [@ locationExp]
JoinStatement ::= join casteName ([params])
QuitStatement ::= quit [ casteName ]
SuspendStatement ::= suspend casteName
ResumeStatement ::= resume casteName
EvolveStatement::=
   evolve [casteName] to casteName ([params])
destroyStatement := destroy [ agentVar ]
```

(3) Event-Driven Programming Facilities

CAOPLE has two statements that support event-driven computing: the WHEN-statement and the TILL-statement. Both test for whether the system is in a scenario, which are conditions on whether an action is performed either by a specific agent or an agent of a certain caste.

```
WhenStatement ::= when scenario { statements } ;
TillStatement ::= till  scenario ;
scenario ::= AgentVar:ActionID([Params])
   |exist AgentVar in CasteName:ActionID([Params])
```

The statements in a WHEN-statement will be executed if the scenario is true and skipped otherwise. The TILL-statement will delay the execution until the scenario becomes true.

(4) Prevention of Data Races

The write-write type of data racing is not possible because the state variables of each agent can only be changed by the agent itself. The write-read type of data racing can be prevented by using the WITH-statement, which has the following syntax:

```
WithStatement::= with var = expr { statements }
```

Here, `expr` is a variable of a structured data type. When the statement is executed, the value of `expr` is copied to a new variable `var` of the same data type, which is then changed by the statements and copied back to `expr` as an atomic operation. Consider the following example:

```
with date= conf.date {
   date.day := 29;
   date.month := 03;
   date.year := 2016;
};
```

This updates *conf.date* from a previous value such as 28/02/2018 to the new value 30/03/2018 in a single atomic operation making it impossible for other agents to read the data when, for example, only the day has been changed.

(5) Example 2: API

In the program below, an agent of caste `RandomIntGenerator` is a service that generates a random integer whenever its requestor asks for it, doing so by performing an observable action that has the random number as a parameter; the details of that action are omitted for the sake of space.

```
uses RandomIntRequestor;
caste RandomIntGenerator(req: RandomIntRequestor) {
   observe RandomIntRequestor;
   var randomInt: int;
   var myRequestor: RandomIntRequestor;
   action RandomIntGenerated(rand: int){
      … /* Details omoitted */
      rand := randomInt;
   }
   init{
      randomInt := 0;
      myRequestor:=req;
   }
   body{
      when myRequestor: RequestRandomInt() {
         RandomIntGenerated(randomInt);
      }
   }
}

uses RandomIntGenerator;
caste RandomIntRequestor(){
   var myGenerator: RandomIntGenerator;
   action RequestRandomInt(){}
   init{
    create myGenerator of RandomIntGenerator(self);
   }
   body{}
}
```

The caste `RandomIntRequestor` can be considered to be an API for using the service `RandomIntGenerator`, because it creates an instance of it upon initialisation and defines an action `RequestRandomInt`, which is taken by the requestor when it requires a random integer. The following caste ServiceRequestor uses the random number generator service by extending the `RandomIntRequestor` caste.

```
uses RandomIntGenerator, RandomIntRequestor;
caste ServiceRequestor() extend RandomIntRequestor {
   observe RandomIntGenerator;
   var randomInt: int;
   action RequestService(){ }
   init {super();}
   body{
      RequestRandomInt();
      till myGenerator:
         RandomIntGenerated(rcv randomInt);
      var job : Job;
      job.content := randomInt;
      RequestService(job);
   }
}
```

(6) Example 3: Elastic Load Balancing

The caste `LoadManager` below implements a load balancer, which receives service requests from agents of caste `ServiceRequestor` and allocates the job to one of its workers (agents of caste `Worker`), which provide the services. A commonly used way of allocating these jobs is the *round robin* algorithm, which assigns jobs to the workers in turn. It is implemented below.

```
import LoadBalancorDefs;
uses ServiceRequestor, Worker;
```

```
caste LoadManager() {
   observe ServiceRequestor, Worker;
   var nWorkers: int;
   var nMachines: int;
   var index: int;
   var jobQueueLength: int;
   var listOfWorkers : ListOfWorkers;
   var listOfMachines: ListOfMachineIPs;
   var worker: Worker;
   action AllocateJob(i: int, j: Job){
      index:= index+1;
      if (index>=nWorkers){index:=0;}
   }
   action stopWorker(i: int){ }
   action AddWorker(){
     var machineIP: string;
     machineIP:=listOfMachines[nWorkers%nMachines];
     create worker of Worker(nWorkers) @ machineIP;
     till worker: iAmReady();
     listOfWorkers[nWorkers] := worker;
     nWorkers:=nWorkers+1;
   }
   action ReduceWorker(){
      nWorkers:=nWorkers-1;
      stopWorker(nWorkers);
   }
   init {
      listOfMachines := … /* initialise the var */
      nMachines :=listOfMachines.length;
      nWorkers:=0;
      for (var j:=0 to 4) { AddWorker(); };
      index:=0;
      jobQueueLength:=0;
   }
   body{
      var job: Job;
      when exist R in ServiceRequestor:
         RequestService(rcv job) {
         AllocateJob(index, job);
         jobQueueLength := jobQueueLength+1;
         if (jobQueueLength / (nWorkers+1) >=10){
            AddWorker();
         };
      };
      when exist W in Worker: JobDone() {
         jobQueueLength := jobQueueLength-1;
         if ((jobQueueLength < nWorkers )
            && (nWorkers >1 ))  {
            ReduceWorker();
         };
      };
   }
}
```

The above load balancer is elastic. The number of unfinished jobs per worker on average is calculated as a measure of the load. When it is greater than a threshold (10), the load balancer will create a new worker to deal with the demand. When it drops to below 1, at least one worker must be idle so it is removed. The actions that implement addition and removal of a worker are `AddWorker` and `ReduceWorker`.

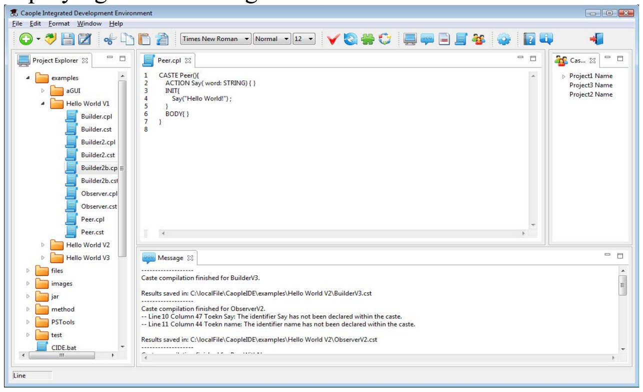
The caste `Worker` below implements the service providers. It defines two actions: `JobDone` for announcing that a job has been finished by the service provider and `iAmReady` for announcing that the service has finished initialisation and is ready to take on jobs.

```
uses LoadManager;
caste Worker(id : int) {
   observe LoadManager;
   var myId: int;
   var workerId: int;
   var job: Job;
   action JobDone(wID: int) { }
   action iAmReady() { }
   init {
      myId:= id;
      iAmReady();
   }
   body{
      var hasNoWorkToDo: Bool;
      hasNoWorkToDo := true;
      when exist B in LoadManager:
         AllocateJob(rcv workerId, rcv job){
         if (workerId == myId) {
            hasNoWorkToDo:= false;
            wait 100;  /* Do job */
            JobDone(myId);
         }
      };
      if (hasNoWorkToDo) {
         when exist B in LoadManager:
            stopWorker(myId){
            destroy;
         }
      }
   }
}
```

Note that when a worker receives the instruction to stop, it will complete the queue of jobs already allocated to it.

B. The Integrated DevOps Environment CIDE

CIDE is an integrated DevOps Environment for the CAOPLE language. Figure 6 is the user interface for editing and compiling CAOPLE programs; there are also tools for deploying and executing code.



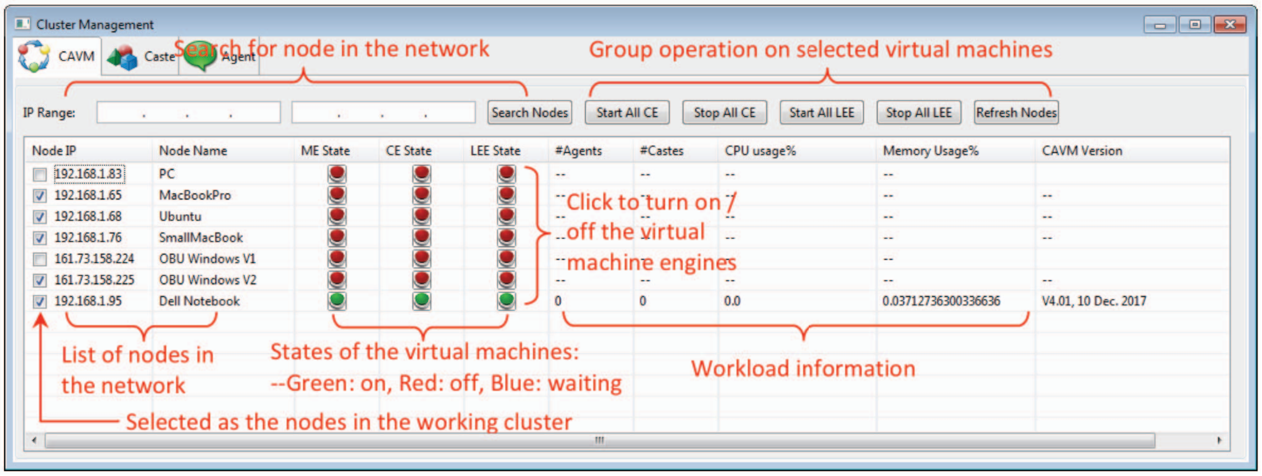
```
CASTE Peer(){
   ACTION Say( word: STRING) { }
   INIT{
      Say("Hello World!") ;
   }
   BODY{ }
}
```

```
------------------
Caste compilation finished for BuilderV3.

Results saved in: C:\localFile\CaopleIDE\examples\Hello World V2\BuilderV3.cst
------------------
Caste compilation finished for ObserverV2.
-- Line 10 Column 47 Toekn Say: The identifier Say has not been declared within the caste.
-- Line 11 Column 44 Toekn name: The identifier name has not been declared within the caste.

Results saved in: C:\localFile\CaopleIDE\examples\Hello World V2\ObserverV2.cst
------------------
```

Figure 6. CIDE's Graphical User Interface for Editing Code

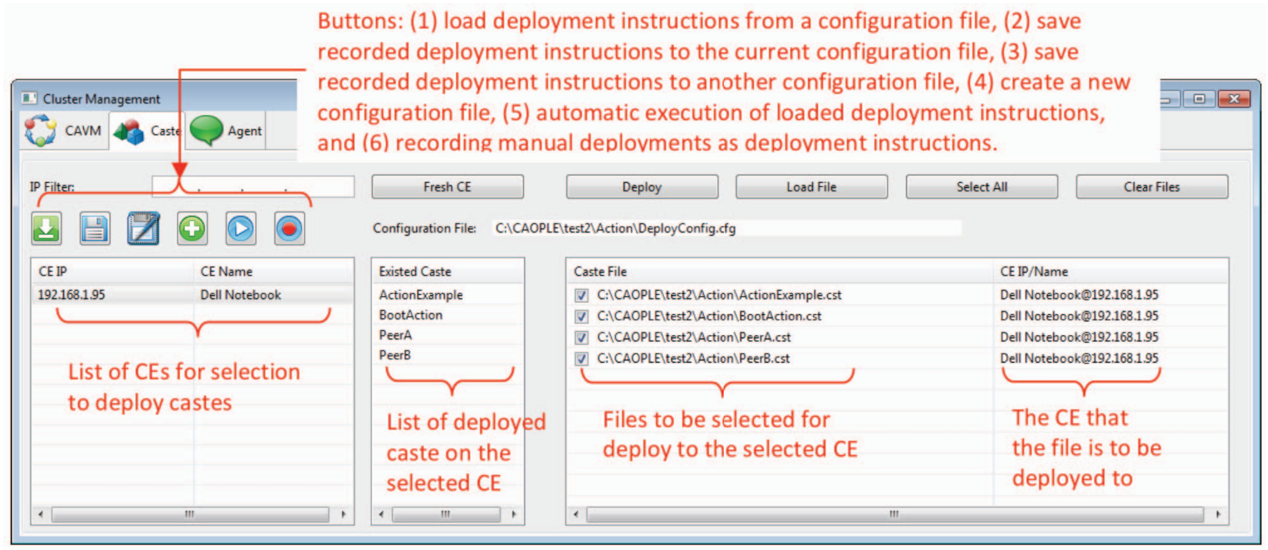
Caste is the unit both of compilation and of deployment. There are no build or link operations. Each machine in a cluster can run either a Communication Engine (CE) or a Logic Execution Engine (LEE) or both, where CE and LEE are two parts of the CAVM-2 virtual machine. The object codes of the castes are deployed to the CEs and the agents (i.e. the instances of the castes) run on the LEEs. Any LEE can be chosen no matter where the object code is deployed. The CE manages communication between agents. Each cluster can have multiple CEs and LEEs.

The user can view the set of nodes in the network and select a subset of them as his/her working cluster as shown in **Error! Reference source not found.**. Each virtual machine on the nodes can be, with a click, switched on (green) or off


Cluster Management
CAVM
Caste
Agent
Search for node in the network
Group operation on selected virtual machines
IP Range:
Search Nodes
Start All CE
Stop All CE
Start All LEE
Stop All LEE
Refresh Nodes
Click to turn on / off the virtual machine engines
List of nodes in the network
States of the virtual machines: --Green: on, Red: off, Blue: waiting
Workload information
Selected as the nodes in the working cluster

	Node IP	Node Name	ME State	CE State	LEE State	#Agents	#Castes	CPU usage%	Memory Usage%	CAVM Version
☐	192.168.1.83	PC	🔴	🔴	🔴	--	--	--	--	
☑	192.168.1.65	MacBookPro	🔴	🔴	🔴	--	--	--	--	--
☑	192.168.1.68	Ubuntu	🔴	🔴	🔴	--	--	--	--	--
☑	192.168.1.76	SmallMacBook	🔴	🔴	🔴	--	--	--	--	--
☐	161.73.158.224	OBU Windows V1	🔴	🔴	🔴	--	--	--	--	
☑	161.73.158.225	OBU Windows V2	🔴	🔴	🔴	--	--	--	--	--
☑	192.168.1.95	Dell Notebook	🟢	🟢	🟢	0	0	0.0	0.03712736300336636	V4.01, 10 Dec. 2017

Figure 8. User Interface for managing cluster.


Buttons: (1) load deployment instructions from a configuration file, (2) save recorded deployment instructions to the current configuration file, (3) save recorded deployment instructions to another configuration file, (4) create a new configuration file, (5) automatic execution of loaded deployment instructions, and (6) recording manual deployments as deployment instructions.
Cluster Management
CAVM
Caste
Agent
IP Filter:
Fresh CE
Deploy
Load File
Select All
Clear Files
List of CEs for selection to deploy castes
List of deployed caste on the selected CE
Files to be selected for deploy to the selected CE
The CE that the file is to be deployed to

Configuration File: C:\CAOPLE\test2\Action\DeployConfig.cfg

CE IP	CE Name
192.168.1.95	Dell Notebook

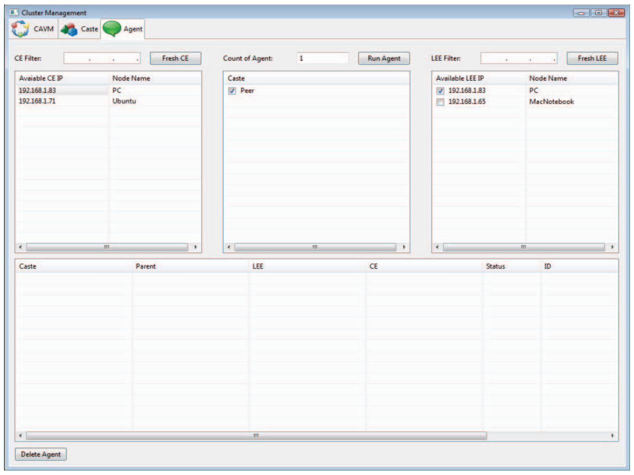
Existed Caste
ActionExample
BootAction
PeerA
PeerB

	Caste File	CE IP/Name
☑	C:\CAOPLE\test2\Action\ActionExample.cst	Dell Notebook@192.168.1.95
☑	C:\CAOPLE\test2\Action\BootAction.cst	Dell Notebook@192.168.1.95
☑	C:\CAOPLE\test2\Action\PeerA.cst	Dell Notebook@192.168.1.95
☑	C:\CAOPLE\test2\Action\PeerB.cst	Dell Notebook@192.168.1.95

Figure 7. CIDE's Caste Management Tool for Deployment of Object Code to Communication Engines

(red). Machines can also be added to or removed from the set. Information about the workloads on the selected machines (such as the usages of CPU and memory, number of agents running on the LEE and the number of castes deployed to the CE) can also be obtained with a click of a button and displayed on screen.

The object code for a caste can be deployed to a CE using the caste tab; see **Error! Reference source not found.**. Manual deployment can be easily performed by selecting a object code file from the machine's file system and selecting the machine the code is to be deployed to and then clicking the deploy button. Manual deployment can be recorded, and saved to a configuration file with a click of button. Previously recorded deployments can be loaded to CIDE and automatically executed when a set of object codes needs to be deployed again after testing and debugging. The list of castes deployed to a CE can also be obtained and displayed on screen.



Count of Agent: 1

Avaiable CE IP	Node Name
192.168.1.83	PC
192.168.1.71	Ubuntu

	Caste
☑	Peer

	Available LEE IP	Node Name
☑	192.168.1.83	PC
☐	192.168.1.65	MacNotebook

Caste	Parent	LEE	CE	Status	ID

Figure 9. CIDE's Agent Management Tool

Once the object code has been deployed to a CE, agents

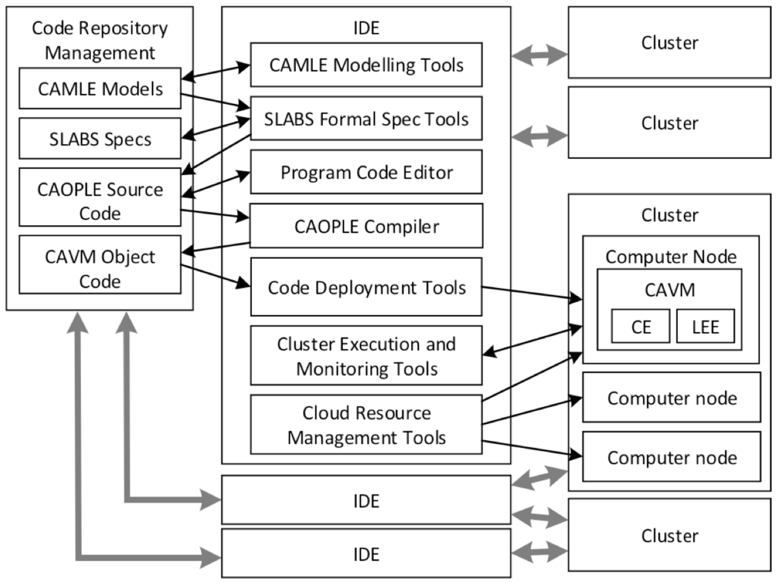


Figure 10. Architecture of CIDE

can be created to run on any machine in the cluster with a couple clicks of buttons using the agent tab; see Figure 9. The running state of each agent on each LEE can also be monitored. A selected running agent can be stopped when needed to with a click of the *Delete Agent* button.

CIDE has been implemented in Java. **Error! Reference source not found.** shows the architecture of CIDE.

C. Summary

CIDE is an Integrated DevOps Environment because it offers code deployment, cluster management and agent management in addition to the features of a traditional Integrated Development Environment. In this way, distributed and parallel programs can be tested and run on a cluster. Uploading a CAVM is all that is needed to install a server.

Note also that CAOPLE programs are "*write-once, run anywhere*". The virtual machine has been tested on Windows, Linux and Mac OS machines. No change is needed to the object code when it is moved onto a different machine so clusters can be heterogeneous.

Finally, note that the object code of a caste is two orders of magnitude smaller than that of container images of Docker, making it possible for the deployment and initialization of an agent to take only a few milliseconds rather than the seconds taken by Docker. The sizes of the object code files for castes are typically a few KBs. Our experience with CAOPLE and CIDE indicate that together they achieve the aims of Docker and container orchestration engines better than Docker does itself. More importantly, testing microservices can be done in a cluster environment as a part of the programming phase.

VI. CONCLUSION

The past few years have witnessed a paradigm shift in the practice of cloud software engineering. There are a number of new fundamental concepts:

- Software applications running on a cloud consist of a large number of autonomous active computational entities called microservices, each wrapped as containers, distributed over a network and executed in parallel. We call them agents. Their communication is asynchronous and non-blocking. The connections are dynamically established.
- Agents are dynamically instantiated from templates, also called microservices but represented as container images. We call these castes.
- Software processes now include development but also deployment and operation. These processes are integrated and pipelined to help deal with the complexity of the infrastructure and environment. Since it must be possible for microservices to be continuously added and removed, the emphasis is on continuous integration, testing and delivery. Both the microservices themselves and the developers maintaining them form an ecosystem.
- Tools exist to make deployment of code, instantiation of agents and monitoring of clusters as efficient as possible.

We argue that a new programming model that directly supports the new paradigm would significantly improve both software quality and productivity. This would necessitate a new programming language for microservices instead of viewing it as simply an architectural style for which any programming language is suitable. Goals of such a new programming language would include:

- Language facilities at a high level of abstraction that match the metaphors of the paradigm;
- Obviating the needs for low level communication primitives or network location sensitivity;
- Code-once-run-anywhere, reducing the complexities of heterogeneous hardware and software platforms;
- Supporting DevOps in one Integrated DevOps Environment;

In this paper, we examined some existing programming models in the light of the emerging paradigm. We argued that service agent is the best conceptual model for a programming language in the new paradigm. We briefly reported both CAOPLE and CIDE. Our preliminary experiments show that both are promising.

For future work, we are searching for a new software development methodology for cloud-based systems and a way to reason about their properties.

REFERENCES

[1] Lewis, J., and Fowler, M., *Microservices*. URL: http: //martinfowler.com/articles/microservices.html#footnote-monolith, 25 Mar. 2014. (*Last access on 2 Nov. 2015*)
[2] NewMan, S., Building Microservices: Designing Fine-Grained Systems. O'Reilly, Feb., 2015.
[3] Krause, L., *Microservices: Patterns and Applications*.

Amazon.co.uk, Marston Gate, April, 2015.

[4] Negus, C., *Docker Containers: Build and Deploy with Kubernetes, Flannel, Cockpit, and Atomic*. Prentice Hall, 2016.

[5] Miell I., and Sayers, A. H., *Docker in Practice*. Manning, 2016.

[6] Sharma S., and Coyne, B., *DevOps for the Dummies*, 2nd IBM Limited Edition. John Wiley & Sons, 2015.

[7] Barendregt, H., *The Lambda Calculus: Its Syntax and Semantics*. College Publications, 2013.

[8] Kuhn, T., *The Structure of Scientific Revolutions*, 2nd, ed. University of Chicago Press, 1970.

[9] Santoli, G., *Microservices Architectures: Become A Unicorn Like Netflix, Twitter And Hailo*. Presentation Slides, Mar 31, 2016. URL: https://www.slideshare. net/gjuljo/microservices-architectures-become-a-unicorn-like-netflix-twitter-and-hailo. (*Last access on 8 May, 2017*)

[10] Mauro, T., *Adopting Microservices at Netflix: Lessons for Architectural Design*. Technical Blog, NGINX, URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/ (*Last access on 8 May 2017*)

[11] Santoli, G., *Microservices Architectures: Become A Unicorn Like Netflix, Twitter And Hailo*. Presentation Slides, Mar 31, 2016. URL: https://www.slideshare. net/gjuljo/microservices-architectures-become-a-unicorn-like-netflix-twitter-and-hailo. (*Last access on 8 May, 2017*)

[12] Munns, C., *Microservices at Amazon*. Slides of presentation at the I-Love-APIs 2015. URL: https://www.slideshare.net/ apigee/i-love-apis-2015-microservices-at-amazon-54487258. (*Last access on 8 May 2017*).

[13] Shoup, R., *From Monolith to Microservices - Lessons from Google and eBay*. Webex recording (slides and audio) of MicroServices Meetup. URL: https://cisco.webex.com/ ciscosales/lsr.php?RCID=8d18be1e6fef4a1dad8b408453a2f66 2 (*Last access on 8 May 2017*)

[14] Microsoft, *Microsoft Azure Service Fabric*. URL: https: //azure.microsoft.com/en-us/ (*Last access on 8 May 2017*)

[15] Smartbear, State of API Report 2016: A Global Survey Looking at the Growth, Opportunities, Challenges & Processes in the API Industry in 2016. Feb 24, 2016. URL: http://blog.smartbear.com/api-testing/api-trends2016/?q= State+of+API+Report#ga=2.42711077.1495982967.1494325 736-971574768. 1494325564. (Last access on 8 May 2017)

[16] Clark, K., J., Microservices, SOA, and APIs: Friends or enemies? A Comparison of Key Integration And Application Architecture Concepts for An Evolving Enterprise. IBM DeveloperWorks, January 21, 2016.

[17] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J. and Josuttis, N., "Microservices in Practice, Part 1: Reality Check and Service Design". *IEEE Software*, Vol. 34, No. 1, pp91-98, Jan.-Feb., 2017.

[18] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., and Josuttis, N., "Microservices in Practice, Part 2: Service Integration and Sustainability", *IEEE Software*, Vol. 34, No.2, pp97-104, Mar.-Apr., 2017.

[19] Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M., *Microservice Architecture: Aligning Principles, Practices, And Culture*. O'Reilly Media, Inc., June 2016.

[20] Daya, S. et al., Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach. IBM Redbooks, IBM, August 2015.

[21] Pahl, C., "Containerization and the PaaS Cloud", *IEEE Cloud Computing*, Vol. 2, No. 3, pp24-31, May-Jun. 2015.

[22] Merkel, D., "Docker: Lightweight Linux Containers for Consistent Development and Deployment". *Linux Journal*, Vol. 2014, No. 239, p2, 2014.

[23] Docker, *What is a Container: A Standardized Unit of Software*. URL: https://www.docker.com/what-container. (Last access on 8 May 2017)

[24] Gupta, A., Docker for Java Developers: Package, Deploy and Scale with Ease. O'Reilley, 2016.

[25] Ismail, U., Comparing Orchestration Engine Options in Rancher. Rancher Labs, Oct. 2016.

[26] Farcic, V., The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices. Leanpub, May 2016.

[27] RightScale, *RightScale 2016 State of the Cloud Report*. URL: http://assets.rightscale.com/uploads/pdfs/rightscale-2016-state-of-the-cloud-report-devops-trends.pdf (Last access on 9 May 2017).

[28] RightScale, *RightScale 2017 State of the Cloud Report*. URL: http://assets.rightscale.com/uploads/pdfs/RightScale-2017-State-of-the-Cloud-Report.pdf. (Last access on 9 May 2017).

[29] Hewitt, C., ActorScript extension of C#, Java, Objective C, JavaScript, and SystemVerilog using iAdaptive concurrency for antiCloud^{TM} privacy and security. URL: https://arxiv.org/ pdf/1008.2748.pdf. (Last access on 21 August 2017).

[30] Donovan, A. A. and Kernighan, B. W., *The Go Programming Language*. Addison-Wesley, 2016.

[31] Odersky, M., Spoon L., and Venners, B., *Programming in Skala*, 3rd Edition. Artima, 2016.

[32] Hewitt, C., Bishop, P., Steiger, R., "A Universal Modular Actor Formalism for Artificial Intelligence". *Proc. of IJCAI'73*, pp235-245, 1973.

[33] Akka: Build powerful reactive, concurrent, and distributed applications more easily. URL: http://akka.io (Last access on 20 August, 2017)

[34] Wikipedia, *Actor model*. URL: https://en.wikipedia.org/wiki/ Actor_model. (*Last access on 22 August 2017*).

[35] Bonér J., and Klang, V., *Reactive programming vs. Reactive systems*. O'Reilly Media, Dec. 2, 2016. URL: https://www. oreilly.com/ideas/reactive-programming-vs-reactive-systems. (*Last access on 17 June 2017*)

[36] Bainomugisha, E., Carreton, A. L., van Cutsem, T., Mostinckx, S., de Meuter, W., "A survey on reactive programming", *ACM Computing Surveys*, Vol. 45 Issue 4, August 2013.

[37] Zhu, H., "Formal Specification of Agent Behaviour through Environment Scenarios", *Formal Aspects of Agent-Based Systems*, Rash, J. et al. (eds.), Springer LNCS 1871, pp263-277, April 2000.

[38] Zhu, H., "SLABS: A Formal Specification Language for Agent-Based Systems", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 5, pp529~558, Nov. 2001.

[39] Xu, C., Zhu, H., Bayley, I., Lightfoot, D., Green, M., and Marshall, P., "CAOPLE: A Programming Language for Microservices SaaS", in *Proc. of SOSE 2016*, pp42-52, April 2016.